MASTER IN INNOVATION AND RESEARCH IN INFORMATICS

Advanced Computing

Master Thesis

Edge crossings in linear arrangements: from theory to algorithms and applications

Author
Lluís Alemany Puig

Advisor
Professor Ramon Ferrer i Cancho
Complexity & Quantitative Linguistics Lab,
Laboratory for Relational Algorithmics, Complexity and Learning (LARCA),
Computer Science Department

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Defense date: July 2019

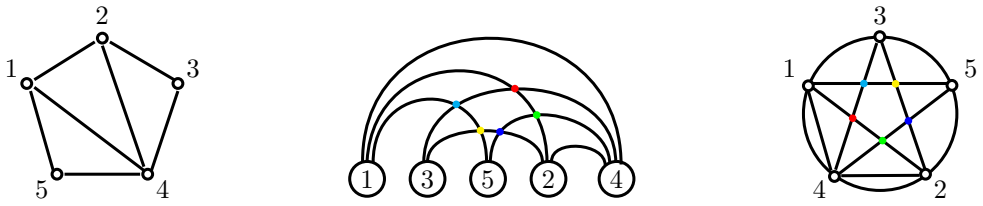
Abstract

In linguistic research, the structure of sentences is often modelled as a tree where vertices are words and edges indicate syntactic dependencies. Interest in the statistical properties of these graphs has been growing. From a graph theoretic standpoint, a sentence is linear arrangement of the vertices of a tree. Here we focus on one feature: the number of edge crossings in linear arrangements. We develop various algorithms to count them and to investigate their properties in random linear arrangements. We show that crossings of edges in linear arrangements of the vertices of a graph can be computed efficiently and provide specialised algorithms for its computation in dense graphs and in trees. We also devise novel algorithms for the exact computation of the variance in randomly uniform linear arrangements of the number of crossings. We give algorithms for general graphs, trees and forests. The last two have linear-time complexity in the number of vertices. Without these algorithms the variance had to be approximated using Monte Carlo procedures whose number of iterations is directly proportional to the $n!$ different linear arrangements of the vertices of a graph. Moreover, we study the distribution of the variance in Erdös-Rényi graphs, and give constant-time algorithms for the prediction of the number of crossings given the lengths of the edges. Finally, we use these algorithms to provide new evidence to support the hypothesis that the scarcity of crossing dependencies is a side-effect of dependency length minimisation. The algorithms devised in this work have been encapsulated in a C++ library, that is to be made publicly available in the near future.

Contents

1 Introduction

The concept of crossing between two edges of a graph has been defined for long, first introduced by Ringel [30] in 1963 and independently by Grünbaum [15] in 1972. In a *topological* setting, the crossing number of a graph G, $cr(G)$, is defined as the minimum number of edge crossings produced over all possible drawings of G where edges are drawn as curves. In a closer setting to ours, we find the *rectilinear* crossing number $\overline{cr}(G)$ which differs from the previous in that edges are forced to be drawn using straight lines. The setting studied in this work is that of the number of crossings in a 1-page drawing of a graph. Informally, edges are placed along a line, in this work called *linear arrangement*, and also known as *spine*, and edges are all drawn above the line. Figure 1.1a shows an example of a labelled graph whose vertices have been arranged in a way that produces 5 crossings (figure 1.1b). In [28, Formulation 3] we find an equivalent formulation of this problem: to arrange the vertices of a graph so that they lie on a circle and their edges are chords of the circle, also known as *convex setting*. In this case, the number of crossings is denoted as $cr^{\circ}(G)$. An example is shown in figure 1.1c. Interestingly, if the vertices are placed along the boundary of a circle and the edges drawn within its interior, the *topological* and *convex* setting are equivalent.



(a) Graph example. (b) A linear arrangement in 1 page. (c) *convex* setting.

Figure 1.1: (1.1a) A graph, (1.1b) a possible linear arrangement of its vertices, and (1.1c) the same arrangement but with the vertices placed on a circle.

Regarding the optimisation of crossings in linear arrangements, one may be interested in finding one that does not produce edge crossings in the k-page setting [28, Formulation 1]. This is the problem of finding a book *embedding* of a graph, where its vertices are placed along the spine of the book (our linear arrangement) and the edges in the same page do not produce crossings. Interestingly, [28, Formulation 3] hints that this problem can be seen as placing the vertices of the graph on a circle, making its edges be chords of the circle, and assigning the edges to layers so that the chords (edges) in the same layer do not cross. Even more interesting, Bernhart *et. al.*[4] showed that all outerplanar graphs are 1-page embeddable. This implies that all trees T admit a linear arrangement for which no pair of edges cross. Moreover, Chung *et. al.*[28], among other contributions, characterised several types of graphs that can be embedded in books with 1 or 2 pages beyond the outerplanar graphs.

Linear arrangements have been used quite often in the literature for optimisation problems. One very-well studied is the Minimum Linear Arrangement problem whose decisional version consists on deciding whether there exists a linear arrangement π of the vertices of a graph G that yields a sum of the length of the edges smaller than a certain constant $l \in \mathbb{N}^{+}$. This problem, originally named Optimal Linear Arrangement, was shown to be NP-Complete by Garey and Johnson [14] for general graphs. In trees, however, the problem becomes polynomial-time solvable and several algorithms have been devised. Shiloach [32] found an $O\left(n^{2.2}\right)$-time algorithm, later corrected by Esteban *et. al.*[11], and Chung [9] devised a better one with cost $O\left(n^{1.585}\right)$. Hochberg *et. al.*[18] gave a linear-time algorithm to minimise the sum of the length of the edges of trees in 1-page embeddings, namely, finding the optimal linear arrangement (one that yields minimum sum of length of edges) is linear-time solvable for trees if we restrict the arrangements to those that do not produce edge crossings. Figure 1.2 gives an example with which it can be seen that the optimal linear arrangement in 1-page embedding need not yield the same sum of lengths as that of an optimal arrangement in a general embedding.

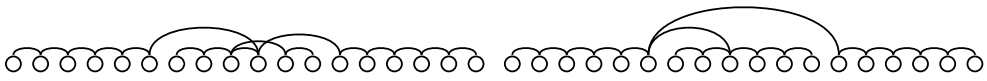


Figure 1.2: Two optimal linear arrangements of the vertices of a graph (left) when allowing crossings, and (right) in a 1-page embedding, i.e., disallowing crossings. Source [18, Figure 1].

Formally, given a simple labelled graph $G = (V, E)$ (which we briefly call *graph*), with $n = |V| = |\{1, \cdots, n\}|$ vertices, $m = |E|$ edges, $A^x = [a_{ij}^{(x)}]$ the x-th power of its adjacency matrix, and maximum degree k_{max}, let a *linear arrangement* of its vertices be a function $\pi : V \to \{1, \cdots, n\}$ that gives the position $1 \leq p \leq n$ of each vertex $u \in V$ in the linear arrangement. We can also see this function as a permutation of the sequence of values from $[n]$. The number of crossings of a graph G in a linear arrangement π is readily defined as the total amount of edges that cross in that linear arrangement

$$C_G(\pi) = \frac{1}{2} \sum_{st \in E} \sum_{uv \in E} c_\pi(st, uv), \tag{1.1}$$

where $c_\pi : E \times E \to \{0, 1\}$ is 0 when the edges $\{s, t\}, \{u, v\} \in E$ do not cross, and is 1 when they do. Two edges $\{s, t\}, \{u, v\} \in E$ cross if, and only if the positions of the vertices are interleaved in the linear arrangement, namely

$$\pi(s) < \pi(u) < \pi(t) < \pi(v), \quad \text{or} \quad \pi(u) < \pi(s) < \pi(v) < \pi(t). \tag{1.2}$$

One of the contributions of our this are efficient algorithms for the computation of the number of crossings in a graph when its vertices are linearly arranged, summarised in table 1.1.

Algorithm	Cost	
	Time	Space
Brute force using Q (section 2, algorithm 2.1)	$\Omega(\|Q\|)$, $O(m^2)$	$O(1)$
Brute force not using Q (section 2, algorithm 2.2)	$\Omega(\|Q\|)$, $O(m^2)$	$O(1)$
Dynamic programming (1) (section 2.1, algorithm 2.5)	$O(n^2)$	$O(n^2)$
Dynamic programming (2) (section 2.2, algorithm 2.6)	$O(n^2)$	$O(n)$
Stack-based (section 2.3, algorithm 2.9)	$O(m \log k_{max})$	$O(m)$
Stack-based (on trees) (section 2.3, algorithm 2.9)	$O(n \log k_{max})$	$O(n)$

Table 1.1: Summary of the algorithms devised for computing $C_G(\pi)$.

Since edges that have common vertices cannot possibly cross, according to the definition of crossing, we can give an alternative definition of the number of crossings in a graph G involving the set of pairs of independent edges $Q(G)$

$$C_G(\pi) = \sum_{\{st, uv\} \in Q(G)} c_\pi(st, uv). \tag{1.3}$$

In [2] was shown that the expectation of this value in random linear arrangements (rla) can be given as a function of the set of pairs of independent edges, $Q(G)$, i.e. the set of pairs of edges that have no common vertices. Then,

$$\mathbb{E}_{rla}[C_G] = \frac{1}{n!} \sum_{\pi \in \Pi(G)} C_G(\pi) = \frac{1}{3}|Q(G)| \tag{1.4}$$

where $\Pi(G)$ denotes the set of all $n!$ linear arrangements of the vertices of G. Since for any graph G we have that [7]

$$|Q(G)| = \frac{1}{2}\left(m(m+1) - \sum_{u \in V} k_u^2\right) = \frac{1}{2}\left(m(m+1) - n\langle k^2 \rangle\right), \tag{1.5}$$

it is quite straightforward to obtain the expectation of C_G of a particular graph by plugging equation 1.5 into equation 1.4. In 1.5, $\langle k^2 \rangle$ denotes the second moment of the degree about zero, which is merely the average square of the degrees of the vertices of the graph, i.e.

$$\langle k^2 \rangle = \frac{1}{n} \sum_{u \in V} k_u^2.$$

Finding an arithmetic expression for the variance of C_G in random linear arrangements, namely $\mathbb{V}_{rla}[C_G]$, however, is quite more challenging. The problem of finding an expression as simple as possible was not studied, to our best knowledge, until Alemany-Puig and Ferrer-i-Cancho did it in [2]. However, the results given are far from simple. The results presented in their work provide two approaches to the computation of the variance. One shows that the variance can be computed by means of subgraph counting, some of these graphs being the cycle graph of 4 vertices, $\mathcal{C}_4$, two disjoint pairs of two linear trees $\mathcal{L}_3 \oplus \mathcal{L}_3$, and $\mathcal{L}_2 \oplus \mathcal{L}_4$, and other 6 similar graphs ($\oplus$ denotes disjoint union of graphs). The other approach provides an arithmetic expression that requires the computation of several summations that iterate over the elements of Q combined with subgraph counting, these graphs being different from those in the first approach. In this work we use the former to study the variance in random graphs $G_{n,p} \in \mathcal{G}_{n,p}$, where $\mathcal{G}_{n,p}$ is the probability space defined for $0 \leq p \leq 1$ where each element $G_{n,p}$ is a graph of n vertices where each of its edges is selected from the complete graph with probability p [5, Section VII]. For the sake of comprehensiveness, we also studied the expectation of the number of crossings in these graphs. Our analyses show that (propositions 3.1 and 4.1)

$$\mathbb{E}_{n,p}\left[\mathbb{E}_{rla}\left[C_{G_{n,p}}\right]\right] = \binom{n}{4} p^2, \quad \mathbb{E}_{n,p}\left[\mathbb{V}_{rla}\left[C_{G_{n,p}}\right]\right] = \frac{1}{15}\binom{n}{4} p^2(1-p)(p(n^2+n-10)+10).$$

We derive efficient algorithms for the exact computation of $\mathbb{V}_{rla}[C_G]$ with the help of the latter approach given in [2]. Table 1.2 summarises our contributions to the exact computation of $\mathbb{V}_{rla}[C_G]$.

Algorithm	Cost	
	Time	Space
Brute force (section 4, algorithm 4.1)	$O\left(\lvert Q\rvert^2\right)$	$O(1)$
General graphs (1) (section 4.3.2, algorithm 4.2)	$O\left(k_{max} n \langle k^2 \rangle\right)$	$O(n)$
General graphs (2) (section 4.3.2, algorithm 4.3)	$O\left(k_{max} n \langle k^2 \rangle\right)$	$O\left(n + \min\left\{\binom{n}{2}, m + n_G(\mathcal{L}_3)\right\}\right)$
Trees (section 4.3.3, algorithm 4.4)	$O(n)$	$O(n)$
Forests (section 4.3.4, algorithm 4.5)	$O(n)$	$O(n)$

Table 1.2: Summary of the algorithms devised for computing $\mathbb{V}_{rla}[C_G]$, classified by the type of graphs. The first three work for all simple graphs, the second-to-last is specialised on trees, and the last on forests. In practice, the algorithm for general graphs (2) is more efficient than (1).

Other properties can be defined on graphs when its vertices are linearly arranged. One example is the length of the edges: each edge has a length that is the number of vertices inbetween the endpoints plus one, calculated as the absolute value of the difference of the positions of the vertices in the given linear arrangement π. In symbols, the length of an edge is a function θ_π

$$\begin{aligned} \theta_\pi \; : \; E \; &\to \; \mathbb{N} \\ \{s,t\} \; &\to \; |\pi(s) - \pi(t)| \end{aligned} \tag{1.6}$$

Using this we can now define the sum of the length of the edges

$$D_G(\pi) = \sum_{st \in E} \theta_\pi(st). \tag{1.7}$$

Ferrer-i-Cancho [7] gave two surprisingly simple arithmetic expressions for the expectation and variance of D_G in random linear arrangements. The expectation is [7, Equation 6]

$$\mathbb{E}_{rla}[D_G] = \frac{n+1}{3}m, \tag{1.8}$$

and the variance is [7, equation 31]

$$\mathbb{V}_{rla}[D_G] = \frac{n+1}{45}\left[m(2(n-1)-m) + \left(\frac{n}{4}-1\right)n\langle k^2\rangle\right]. \tag{1.9}$$

Surprisingly, as shown by Ferrer-i-Cancho [6], the amount of crossings in a linear arrangement can be predicted when given the lengths of the edges in that arrangement. In this work we give constant-time algorithms, in section 3.2, to calculate $\Pr[\mathit{crossing} \mid d_1, d_2]$, the probability that two edges of length d_1 and d_2 cross when placed uniformly at random in a linear arrangement, which is found at the core of the prediction of C_G (equation 3.5).

Section 5 is the pinnacle of our work. In this section the reader will find an application within the framework of Quantitative Linguistics, a field in which researchers study properties of linguistic networks, where we apply all the algorithms devised in this work. Generally speaking, these networks are graphs where the vertices represent linguistic units (e.g., words) and edges model linguistic relationships between pairs of these units (e.g., syntactic dependencies). Figure 1.4(left) shows an example where there are no crossings, and 1.4(right) shows a variant with one dependency crossing. Our work, done from a purely graph theoretical point of view, allows researchers in this field to study a hypothesis around which we find a heated debate. The hypothesis reads as "uncrossing dependencies could be a side-effect of dependency length minimisation" [21, page 227]. A necessary condition for it to be true, and a weaker version of this hypothesis is that there must exist a positive correlation between C and D [13]. The hypothesis is not always true, since there are cases where minimum D does not yield $C = 0$ [21]. See figures 1.3(left) and 1.3(right) for two examples of sentences where a lower D makes $C > 0$. However, the algorithms devised allowed us to analyse corpus of languages (syntactic dependency treebanks) and extracted empirical evidence that hint the existence of such correlation hence providing evidence to support the weaker hypothesis. In particular, we found that there is a strong positive correlation (Kendall's correlation) between C and D and, more precisely, a strong linear relationship (Pearson's correlation) in all languages of the Universal 2.3 [24], Prague [16], and Stanford Dependencies [20], datasets. Proving causality, and hence the actual hypothesis, is beyond the scope of this work.

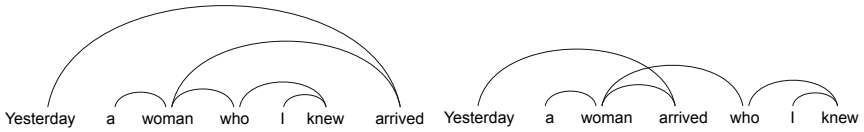


Figure 1.3: Two examples of a linguistic network. Edges represent a syntactic dependency between words. (left) Obviously $C = 0$, and $D = 15$. (right) $C = 1$ and $D = 10$. Source [21, Figure 5].

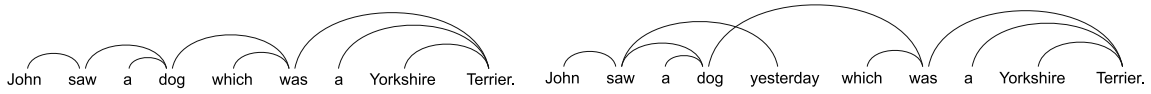


Figure 1.4: Two examples of linguistic networks where (top) there are no dependency crossings, and (bottom) where there is one, due to the introduction of the word "yesterday". Source [21, Figure 1].

Furthermore, all the algorithms devised have been encapsulated in a library so that they are readily available to everyone that might find an interest in them. Such library is briefly described in section A, where we list the algorithms implemented and how they have been tested during their development.

2 Computing the number of crossings

In this section we provide algorithms to efficiently calculate the number of crossings among the edges of a graph in a given linear arrangement π, namely $C_G(\pi)$. First we give two brute force algorithms that introduce the reader to the algorithmic problem, and are crucial for the evaluation of new, more efficient algorithms, i.e, they are used for comparison purposes. Then, in sections 2.1, 2.2 and 2.3 we derive these efficient algorithms and in section 2.4 we compare them.

As explained in the introduction, computing $C_G(\pi)$ can be done in $\Omega(|Q|)$ by implementing equation 1.3 literally, i.e., by enumerating all pairs of independent edges and checking for crossing. It is briefly outlined in pseudocode 2.1.

Algorithm 2.1: Brute-force algorithm to calculate $C_G(\pi)$ in time $O\left(m^2\right)$, using the set $Q(G)$.

Input: $G = (V, E)$ a graph, π a linear arrangement of the vertices.
Output: $C_G(\pi)$, the total number of crossings.

```
1 Function Crossings-BruteForce-Q(G, π) is
2     C ← 0 the number of crossings
3     for {st, uv} ∈ Q(G) do
4         C ← C + c_π(st, uv)
5     return C
```

This constitutes a brute force algorithm whose cost turns $O\left(m^2\right)$ since $|Q|$ approaches $\binom{m}{2}$. There is another brute force algorithm for $C_G(\pi)$, slightly more efficient in practice but whose cost also tends to $O\left(m^2\right)$. This new procedure is based on a simple equivalent formulation of the problem

$$C_G(\pi) = \sum_{i=1}^{n} \sum_{\substack{v \in V: \\ \{v, \pi^{-1}(i)\} \in E, \\ i < \pi(v)}} \sum_{j=i+1}^{\pi(v)-1} \sum_{\substack{w \in V: \\ \{w, \pi^{-1}(j)\} \in E, \\ \pi(v) < \pi(w)}} 1. \tag{2.1}$$

The first two summations iterate over the set of edges, making sure that none of them are repeated (by using the order of its vertices in π). The first vertex $u = \pi^{-1}(i)$ has position i and the second is vertex v at position $\pi(v)$. We only consider those vertices v adjacent to u such that are to the right of u in π: $i < \pi(v)$. The third summation iterates over all vertices strictly between u and v in π, partially satisfying the condition of crossing in equation 1.2. The inner-most summation iterates over all vertices z adjacent to $w = \pi^{-1}(j)$ such that they are "to the right" of v in π, formally, $\pi(v) < j$. This satisfies completely the condition in equation 1.2. Figure 2.1 illustrates the two cases of two edges crossing and not crossing.

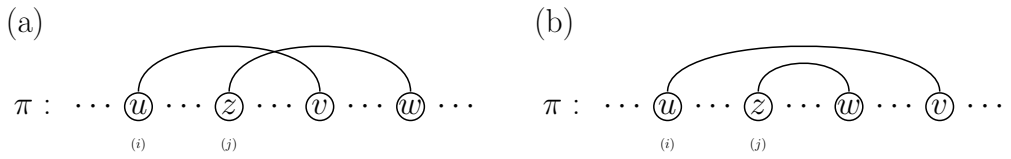


Figure 2.1: Illustration of the brute force algorithm. Given three vertices u, v, and z, all edges intersecting $\{u, v\} \in E$ are those (a) that have a vertex w to the right of v connected to z. (b) Any neighbour of z to the left of v can not cross $\{u, v\}$.

Algorithm in pseudocode 2.2 implements in a straightforward manner equation 2.1.

Algorithm 2.2: Brute-force algorithm to calculate $C_G(\pi)$ in time $O\left(m^2\right)$, without $Q(G)$.

Input: $G = (V, E)$ a graph, π a linear arrangement of the vertices.
Output: $C_G(\pi)$, the total number of crossings.

```
1  Function CROSSINGS-BRUTEFORCE(G, π) is
2      C ← 0 // The number of crossings
3      for i ∈ [1, n] do
4          u ← π^{-1}(i)
5          for v ∈ Γ(u) : π(u) < π(v) do
6              for j ∈ [i + 1, π(v) − 1] do
                   // Iterate through the vertices between u and
                   // v in the linear arrangement.
7                  z ← π^{-1}(j)
8                  for w ∈ Γ(z) : π(v) < π(z) do
9                      if π(u) < π(z) < π(v) < π(w) then
10                         C ← C + 1
11     return C
```

However, as stated at the beginning, this algorithm's cost tends to $\Omega\left(|Q|\right)$, since we are trying to enumerate the pairs of edges that are more likely to cross. When all pairs of edges in Q cross, the bound is tight. In this section we explore other, more efficient algorithms to compute $C_G(\pi)$ by means of counting, and not by enumeration, all summarised in table 1.1. Notice that algorithms 2.1 and 2.2 count the amount of crossings in a way that could be argued that they are actually enumerating them one by one. The coming algorithms tackle the problem avoiding this, namely they count the amount of crossings in bulk, without being able to enumerate them, making them more efficient. The reader is suggested to bear in mind the example in figure 1.1 as we use it as a running example to illustrate the behaviour of the algorithms.

As it is seen in section 2.4, where we compare the performance of our C++ implementations of these algorithms, the third algorithm (with costs $O\left(n^2\right)$, $O\left(n\right)$) appears to be faster for dense graphs than the third (with costs $O\left(m \log k_{max}\right)$, $O\left(m\right)$). However, the third seems to be the fastest for large trees. Also, our claim that algorithm in pseudocode 2.2 is more efficient than a literal implementation of equation 1.3.

2.1 Dynamic programming algorithm (1)

In this section is shown how $C_G(\pi)$ can be computed in $O\left(n^2\right)$ time and $O\left(n^2\right)$ space. This is achieved by means of using two matrices $M, K \in \mathbb{N}^{n \times n}$, which store partial results of the computation of $C_G(\pi)$. Before explaining this algorithm the reader is suggested to take a look at algorithm 2.2, which, although being another brute-force algorithm, serves as a basis for the derivation of the dynamic programming algorithm presented later in this section. Also, in it is used a notation that will prove useful: we denote $\pi^{-1}(p)$ as the vertex in position p in the linear arrangement π. This is trivially computed as an array of n elements in $O\left(n\right)$-time.

As for the dynamic programming algorithm we first start with the definition of matrix M, and we give an $O\left(n^2\right)$-time algorithm to fill it given any linear arrangement π. Then we proceed with matrix K and, finally, we given the $O\left(n^2\right)$-time algorithm to compute $C_G(\pi)$ using K.

Definition and computation of M An intuition for the need of such a matrix is the following: the inner-most loop of algorithm 2.2 (line 8), which implements the inner-most summation of equation 2.1, stores the amount of neighbours a vertex w between u and v in π ($\pi(u) < \pi(w) < \pi(v)$) has strictly "to the right" of v in π. For example, if we choose vertices $u = 1$ and $v = 5$ from figure 1.1b, we can

see that the amount of neighbours of vertex $w = 3$ strictly "to the right" of v are two. This means that, at least, two edges cross the edge $\{u, v\} = \{1, 5\}$ in π. Matrix M allows us to retrieve in constant time this amount every time it is needed.

Matrix M is formally defined as follows: let $u \in V$, and π be a linear arrangement of the vertices in V. Each position in the matrix $M_{j,i}$ is defined, $\forall i, j \ : \ 1 \leq i, j \leq n$, as:

$$M_{i,j} = |\{v \in \Gamma(\pi^{-1}(i)) \mid \pi(v) \geq j\}|. \tag{2.2}$$

In words, $M_{i,j}$ contains the amount of vertices $v \in V$ neighbours of vertex in position i, $\pi^{-1}(i)$, and such that they are "to the right" of position j, formally $\pi(v) \geq j$. This is equivalent to say that $M_{i,j}$ contains the amount of neighbours of $\pi^{-1}(i)$ within the interval $[j, n]$. We can compute all the values of matrix M in $O\left(n^2\right)$ time using algorithm 2.3.

Algorithm 2.3: Calculate M.

Input: $G = (V, E)$ a graph, π a linear arrangement of the vertices.
Output: M, the matrix defined in equation (2.2).

```
1  Function ComputeM(G, π) is
2      M ← n × n matrix
3      for i ∈ [1, n] do
4          d ← k_{π^{-1}(i)}
5          M_{i,1} ← d
6          for j ∈ [2, n] do
7              if {π^{-1}(i), π^{-1}(j − 1)} ∈ E then
8                  d ← d − 1
9              M_{i,j} ← d
10     return M
```

Figure 2.3a gives an example of the matrix M given a linear arrangement of the vertices in the graph example in figure 1.1. Now follows the complexity of this algorithm.

Proposition 2.1. *Algorithm 2.3 has time complexity $O\left(n^2\right)$ and space complexity $O\left(n^2\right)$. It computes the contents of M, as defined in equation 2.2 correctly.*

Proof. It is trivial to see that M is computed correctly. Now, for the time complexity, it is important to take notice on the operation in the conditional in line 7. The complexity of checking the existence of an edge $\{u, v\}$ in a graph depends on its implementation: when using adjacency lists this has time complexity $O\left(\min\{k_u, k_v\}\right)$ and space complexity $O\left(1\right)$, while when using an adjacency matrix, this has time complexity $O\left(1\right)$ and space complexity $O\left(n^2\right)$. Since we are aiming at fast and practical algorithms and since the space complexity will not become any worse, it is suggested to use the adjacency matrix abstraction for this part. Notice, however, that only the row corresponding to vertex $\pi^{-1}(j)$ is actually needed, leading to space $O\left(n\right)$. □

Now, the complexity of algorithm 2.2 can be immediately reduced to $O\left(n^3\right)$ by replacing that inner-most loop (line 8) by accumulating to variable C, the number of crossings, the value $M_{j,\pi(v)+1}$, when $\pi(v) + 1 \leq n$. This is correct since the value in $M_{j,\pi(v)+1}$ is exactly the amount of neighbours of vertex $w = \pi^{-1}(j)$ whose positions lie in the range $[\pi(v) + 1, n]$, which are the only ones that can cross the edge $\{u, v\}$ in π. With this change the space complexity of that algorithm becomes $O\left(n^2\right)$, due to the size of M.

Definition and computation of K Another major improvement can be made in the time complexity by following similar ideas as in the previous section. This time we aim at computing another matrix with which we replace the third summation of equation 2.1, with the help of matrix M. Let

K be a matrix such that in position $K_{i,j}$, $1 \leq i, j \leq n$, is stored the number of potential edges $\{w, z\}$ that would cross an edge connecting $\pi^{-1}(i)$ and $\pi^{-1}(j)$ only when $i < \pi(w) < j < \pi(z)$. We say "potential" because the vertices $\pi^{-1}(i)$ and $\pi^{-1}(j)$ may not be connected. In other words, $K_{i,j}$ contains the amount of vertices to the right of position j (not included) that have a neighbour placed between i and j (also not included). Figure 2.2 illustrates this. Therefore,

$$\begin{aligned} K_{i,j} &= 0, && \forall i, j \ : \ 1 \leq j \leq i+1 < n \\ K_{i,n} &= 0, && \forall i \ : \ 1 \leq n \\ K_{i,j} &= \sum_{k=i+1}^{j-1} M_{k,j+1}, && \forall i, j \ : \ 1 \leq i+2 < j < n. \end{aligned} \tag{2.3}$$

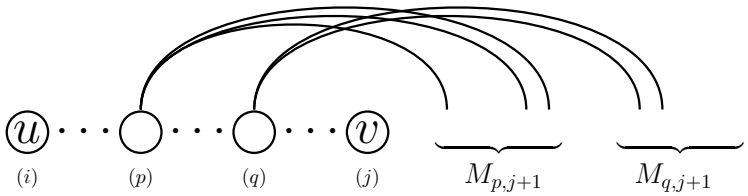


Figure 2.2: Illustration of the contents of $K_{i,j}$. As its definition in equation 2.3 reads, it is equal to $K_{i,j} = \cdots + M_{p,j+1} + \cdots + M_{q,j+1} + \cdots$.

The cases where $K_{i,j} = 0$ are specified to avoid unnecessary computations. Notice that the matrix is symmetric and we only need the upper triangle. Moreover, by definition, those values $K_{i,n}$ are 0 because the edges $\{u, \pi^{-1}(n)\}, \forall u \in V$ can only be crossed by those edges $\{w, z\}$ such that $\pi(w) < \pi(u) < \pi(z) < n$.

The definition of K in equation 2.3 yields an immediate $O\left(n^3\right)$-time algorithm. Fortunately, we can obtain an equivalent definition that takes us closer to our first goal:

Proposition 2.2. *Given a matrix M as defined in equation 2.2, the definition of matrix K in equation 2.3 is equivalent to:*

$$\begin{aligned} K_{i,j} &= 0, && \forall i, j \ : \ 1 \leq j \leq i+1 < n \\ K_{i,n} &= 0, && \forall i \ : \ 1 \leq n \\ K_{i,j} &= M_{i+1,j+1} + K_{i+1,j}, && \forall i, j \ : \ 1 \leq i+2 < j < n. \end{aligned} \tag{2.4}$$

Proof. The first two cases are exactly the same. The proof of the third case is easy.

$$K_{i,j} = \sum_{k=i+1}^{j-1} M_{k,j+1} = M_{i+1,j+1} + \sum_{k=i+2}^{j-1} M_{k,j+1} = M_{i+1,j+1} + K_{i+1,j}.$$

□

Since this procedure might be a bit difficult to understand completely at a first glance, figure 2.3 gives an example of the resulting matrices M and K, given a linear arrangement of a graph. The $O\left(n^2\right)$-time algorithm is given in 2.4.

		1	2	3	4	5
(1)	1	3	3	3	2	1
(3)	2	2	2	2	2	1
(5)	3	2	1	1	1	1
(2)	4	3	2	1	1	1
(4)	5	4	3	2	1	0

(a) Matrix M.

		(0) 1	(2) 2	(4) 3	(1) 4	(3) 5
(1)	1	0	0	2	2	0
(3)	2	0	0	0	1	0
(5)	3	0	0	0	0	0
(2)	4	0	0	0	0	0
(4)	5	0	0	0	0	0

(b) Matrix K.

Figure 2.3: An example of the matrices M and K given the example in figure 1.1. Vertices' indices are indicated between parentheses.

Algorithm 2.4: Calculate K.

Input: $G = (V, E)$ a graph, π a linear arrangement of the vertices.
Output: K, the matrix defined in equation (2.4).

```
Function ComputeK(G, π) is
    M ← ComputeM(G, π) // Use algorithm 2.3.
    K ← n × n matrix initialised at 0
    for i from n − 3 to 1 do
        for j from i + 2 to n − 1 do
            // Thanks to result in proposition 2.2.
            K_{i,j} ← M_{i+1,j+1} + K_{i+1,j}
    return K
```

Proposition 2.3. *Algorithm 2.4 has time complexity $O\left(n^2\right)$ and space complexity $O\left(n^2\right)$.*

Computation of $C_G(\pi)$ Finally, we give the algorithm to compute $C_G(\pi)$ in pseudocode 2.5. This algorithm is fundamentally based on the definition of matrix K (see equation 2.3). For any edge $\{u, v\} \in E$, K stores the number of edges crossing it in $K_{\pi(u),\pi(v)}$. Therefore, after computing matrix K, we only need to iterate over the edges of the graph and accumulate the appropriate values from K.

Algorithm 2.5: Algorithm to compute $C_G(\pi)$ in time and space $O\left(n^2\right)$.

Input: $G = (V, E)$ a graph, π a linear arrangement of the vertices.
Output: $C_G(\pi)$, the total number of crossings.

```
Function ComputeCrossings(G, π) is
    K ← ComputeK(G, π) // Use algorithm 2.4.
    C ← 0 the number of crossings
    for u ∈ V do
        for v ∈ Γ(u) do
            if π(u) < π(v) then C ← C + K_{π(u),π(v)}
    return C
```

Proposition 2.4. *Given a graph $G = (V, E)$ and a linear arrangement π, algorithm 2.5 computes $C_G(\pi)$. It does so in time and space complexity $O\left(n^2\right)$.*

Proof. It is clear from the explanation and previous propositions that the algorithm computes $C_G(\pi)$ correctly. Also, it is easy to see that the time complexity of this algorithm is $O\left(n^2 + m\right) = O\left(n^2\right)$, and uses $O\left(n^2\right)$ space since that is the size of the matrices K and M. □

Further improvements The definition of matrix K (see 2.4) immediately shows that we only require part of it: we only need the range of values $K_{i,j}$, for which $1 \leq i \leq n-3,\ 3 \leq j \leq n-1$. This should be considered when implementing this algorithm since it reduces the space to $(n-3)^2$. Moreover, matrix M can also be reduced to the same size since only these values are actually required, hence reducing the amount of time required for its computation in algorithm 2.3.

2.2 Dynamic programming algorithm (2)

This section describes another dynamic programming algorithm also with time complexity $O\left(n^2\right)$ but with space complexity $O\left(n\right)$. The core idea is to discover edge crossings that produce the edges incident to a vertex $u \in V$ in π when traversing the linear arrangement from position $\pi(u)$ to position n. The edge crossings are discovered when this traversal finds a vertex $v \in V$ that is a neighbour of u, $\{u, v\} \in E$. This traversal is done for every vertex of the graph and in the order they appear in π.

Assume that at iteration i the algorithm is processing vertex $u = \pi^{-1}(i)$. At this iteration we aim at discovering the crossings of the edges between the edge connecting u and one of its neighbours v, such that $\pi(u) < \pi(v)$, and the edges incident to vertices between u and v with one endpoint not between u and v. For this it uses two arrays L_1 and L_2 with a common definition. The algorithm (see pseudocode 2.6) is designed so that at the end of iteration i both L_1 and L_2 contain $L^{(i)}$. It iterates over the vertices of the graph in the order they appear in π (line 4), and uses L_1 to discover edge crossings (line 12) while updating L_2 for later discoveries (line 10). Since at the end of iteration i the two arrays have the same contents, we use $L^{(i)}$ to refer to both lists at the end of said iteration. $L^{(i)}$ is formally defined as

$$
\begin{aligned}
L^{(i)} \in \mathbb{N}^n \ : L^{(0)} &= 0^n, \\
L_p^{(i)} &= L_p^{(i-1)}, && \forall p \in [1, i] \\
L_p^{(i)} &= L_p^{(i-1)} + a_{\pi(i),\pi(p)}, && \forall p \in [i+1, n] \\
\forall i \ : \ 1 \leq i \leq n. &
\end{aligned}
\tag{2.5}
$$

In words, $L_p^{(i)}$ contains the amount of vertices that are neighbours of $\pi^{-1}(p)$ and such that their position is less than or equal to i. One could imagine this as the amount of edges that "stab" vertex $\pi^{-1}(p)$ and that have the other endpoint at a position to the left of i.

Therefore, at iteration i we iterate over the vertices $v = \pi^{-1}(p)$ "to the right" of $u = \pi^{-1}(i)$. When u and v are adjacent (when $a_{\pi(i),\pi(p)} = a_{uv} = 1$) there are two things to be done: (1) update the contents of L_2 so that at the end of that iteration it contains $L^{(i)}$, that is, increment the corresponding position's value by 1 (line 10), and (2) compute the number of crossings that the edge $\{u, v\}$ produces, if any (line 11). In other words, the traversal of the vertices v in π to the right of $u = \pi^{-1}(i)$ discovers crossings between edge $\{u, v\} \in E$ and the edges incident to the vertices strictly between u and v that have one endpoint w such that $\pi(w) < \pi(u)$ (figure 2.4). The main goal of this second traversal is to discover these crossings in constant time. This is done by means of a variable Σ that is updated at every step of the second loop (line 12).

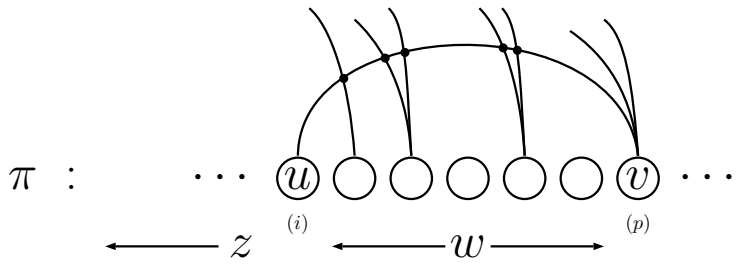


Figure 2.4: When at iteration i, we have that $L_p^{(i)} \geq 5$. We use an inequality since it may have been modified in previous iterations hence containing a larger value. The equality would hold if $i = 1$. At this vertex we discover the amount of edges $\{z, w\}$ that cross $\{u, v\}$. These edges satisfy the definition of crossing $\pi(z) < \pi(u) < \pi(w) < \pi(v)$.

The pseudocode is presented in algorithm 2.6 and its correctness and complexity are given in proposition 2.5.

Algorithm 2.6: Algorithm to compute $C_G(\pi)$ in time $O\left(n^2\right)$ and space $O\left(n\right)$.

Input: $G = (V, E)$ a graph, π a linear arrangement of the vertices.
Output: $C_G(\pi)$, the total number of crossings.

```
1  Function ComputeC(G, π) is
2      C ← 0
3      L_1 ← 0^n, L_2 ← 0^n
4      for i from 1 to n do
5          u ← π^{-1}(i)
6          Σ_i ← 0
7          for p from i + 1 to n do
8              v ← π^{-1}(p)
9              if {u, v} ∈ E then
10                 L_{2,p} ← L_{2,p} + 1
11                 C ← C + Σ_i
12             Σ_i ← Σ_i + L_{1,p}
13         L_1 ← L_2
14     return C
```

Proposition 2.5. *Given a graph $G = (V, E)$ and a linear arrangement π, algorithm 2.6 computes $C_G(\pi)$. It does so in time $O\left(n^2\right)$ and space $O\left(n\right)$.*

Proof. First, we define variable Σ_i (in line 6) at iteration i, $1 \leq i \leq n$ of the first loop (line 4). At iteration p, $i < p \leq n$ of the second loop (line 7), Σ_i contains the amount of edges "stabbing" the vertices strictly between $u = \pi^{-1}(i)$ and $v = \pi^{-1}(p)$ in π. Formally,

$$\Sigma_i = \sum_{l=i+1}^{p-1} L_{1,l}.$$

It is easy to see that variable Σ_i is always updated to follow this definition. Assuming that L_1 is updated correctly to contain $L^{(i-1)}$ at iteration i, whenever u and v are adjacent Σ_i encodes the amount of edges that cross $\{u, v\} \in E$. Now we explain why. Recall the definition of what a crossing is in equation 1.2. Firstly, all edges "stabbing" the vertices $w \in V$ strictly between u and v have their other endpoint at a vertex $z \in V$ such that $\pi(z) < \pi(u)$. This is true by definition of $L^{(i)}$. Secondly,

we know, by construction, that $\pi(u) < \pi(w) < \pi(v)$. Putting everything together, we can see that the condition for crossing holds: $\pi(z) < \pi(u) < \pi(w) < \pi(v)$. See figure 2.4 for an illustration. The edges "stabbing" vertex v should not be accumulated to Σ_i at the moment of iterating over position p. This explains why it is modified at the end of the second loop (line 12). Finally, notice that if $p - i = 1$ then $\Sigma_i = 0$ and no crossings are discovered.

Arrays L_1 and L_2 are updated correctly. The proof is done by induction on i. At the beginning of iteration $i = 1$ of the first loop (line 4) L_1 and L_2 both contain $L^{(0)}$. At the end of the first iteration, just right before updating L_1 in line 13, L_2 contains $L^{(1)}$ since it is correctly updated in line 10: only the positions corresponding to the neighbours $v \in V$ of vertex $u = \pi^{-1}(i)$ such that $\pi(u) < \pi(v)$ are updated. Updating L_1 in line 13 makes both array contain $L^{(1)}$ at the beginning of iteration 2.

The reasoning for any other iteration is similar. Let $i > 1$. Assume that at the beginning of iteration i both L_1 and L_2 contain $L^{(i-1)}$. Similarly, since L_2 is updated correctly, for the same reason explained in the previous paragraph, at the end of that iteration L_2 contains $L^{(i)}$. The update of L_1 in line 13 makes both lists contain $L^{(i)}$ at the beginning of iteration $i + 1$.

It is trivial to see that the time complexity is $O\left(n^2\right)$. As for the space complexity, one could use, for every iteration of the first loop (line 4), a linear amount of space to store the i-th row of the graph's adjacency matrix. This data is constructed in linear time in the degree of that vertex. □

In figure 2.5 is depicted the evolution of the contents of L_1 and L_2 when executing the algorithm on the example of figure 1.1.

i	1	3	5	2	4					
0	0	0	0	0	0	$L_2^{(0)}$	$L_1^{(0)}$			
1	–	0	$1^{\varnothing}$	$1^{\varnothing}$	$1^{\varnothing}$		$L_2^{(1)}$	$L_1^{(1)}$		
2	–	–	$1^{\dagger}$	2^{+}	2^{++}			$L_2^{(2)}$	$L_1^{(2)}$	
3	–	–	–	$2^{\varnothing}$	3^{++}				$L_2^{(3)}$	$L_1^{(3)}$

Figure 2.5: Progress of algorithm 2.6 with input the example in figure 1.1. Column i indicates iteration of the first loop, and the first row contains the vertices of the arrangement. "†" indicates non-adjacent vertices, "∅" indicates no edge crossings were discovered, and "–" indicates "not processed". Each "+" indicates one crossing discovered. To the right is indicated the contents of arrays L_1 and L_2 before the update in line 13: the contents of $L_2^{(0)}$ are first copied into $L_1^{(0)}$, in the next iteration $L_2^{(1)}$ is used and modified, to be ultimately copied into $L_1^{(1)}$, and so on.

2.3 Stack-based algorithm

In this section we describe an algorithm that was originally devised by Kosmas Palios and Georgios Pitsiladis, but that was not properly implemented until now. This algorithm solves the problem with a lower time complexity by using a particular data structure. As before, it also aims at discovering groups of crossings between edges while traversing the linear arrangement. In order to do so, it utilises a stack to which edges are inserted always at the top of it, but that allows for removals at arbitrary positions. First, we present a basic algorithm using a stack. And later, in a more efficient implementation, we replace this stack by a self-balancing binary search tree.

We now give some definitions and introduce the notation used to formalise the algorithm. Here we represent edges as ordered pairs (s, t) where $\pi(s) < \pi(t)$, and call s and t its leftmost and rightmost vertices respectively. Notice that this order depends on the input linear arrangement π. We say that an edge enters (leaves) node u when its rightmost (leftmost) vertex coincides with u. Let Γ_u^+ be the set of entering vertex u, and Γ_u^- be the set of edges leaving u. Then, for all $u \in V$ we have $\Gamma(u) = \Gamma_u^+ \cup \Gamma_u^-$, and, obviously $\Gamma_u^+ \cap \Gamma_u^- = \emptyset$.

In this algorithm we process the vertices of the graph following the ordering in the arrangement from left to right. The core idea is, for every vertex u, to remove from the stack the edges in Γ_u^+,

in increasing edge length order, and count, for every edge removal, the amount of elements on top of the removed edge in the stack. After this step, we insert all edges in Γ_u^-, this time in decreasing edge length. A key fact that guarantees that this algorithm is correct is that every edge is guaranteed to cross all the edges on top of it in the stack at the time of being removed.

We illustrate now how this algorithm works by means of an example, depicted in figure 2.6. Recall the graph and linear arrangement in figure 1.1. We scan the linear arrangement from left to right, in this case starting at vertex 1. Since there are no edges in Γ_1^+ then we simply push into the stack all the edges in Γ_1^- in decreasing edge length. Each edge is assigned an insertion index at the time it is pushed into the stack, which is equal to the amount of edges pushed into the stack so far. Now we repeat the same process for the next vertex. The contents of the stack at this step are depicted in figure 2.6(a). The edges inserted have been assigned the integers from 1 to 5 because we inserted 5 edges. Now we process vertex 5 (figure 2.6(b)), and we find one edge in Γ_5^+ which has to be removed. It was inserted with index 3 and has 2 edges on top of it. This means we have discovered two crossings: the crossings between $(1,5)$ and $(3,4)$, and $(1,5)$ and $(3,2)$. Edge $(1,5)$ is now removed and we insert the only edge in Γ_5^-. The result is depicted in figure 2.6(b). We continue processing the linear arrangement and reach vertex 2. There are two edges in Γ_2^+. When removing them in increasing edge length order, we discover, in total, 3 edge crossings: between the pairs $(3,2)$ and $(5,4)$ (see figure 2.6(c.1)), $(1,2)$ and $(3,4)$, and $(1,2)$ and $(5,4)$ (see figure 2.6(c.2)). We finish processing node 2 by inserting the only edge in Γ_2^-. When processing node 4 we do not discover crossings and we finish the algorithm with an empty stack at the end.

(a)	(b)	(c.1)	(c.2)
5. $(3,2)$	6. $(5,4)$		
4. $(3,4)$	5. $(3,2)$	6. $(5,4)$	
3. $(1,5)$	4. $(3,4)$	4. $(3,4)$	6. $(5,4)$
2. $(1,2)$	2. $(1,2)$	2. $(1,2)$	4. $(3,4)$
1. $(1,4)$	1. $(1,4)$	1. $(1,4)$	1. $(1,4)$
Process vertices 1 and 3.	Process vertex 5. Delete 3. $(1,5)$, insert 6. $(5,4)$.	Process vertex 2. Delete first 5. $(3,2)$, and then 2. $(1,2)$.	

Figure 2.6: Some steps of the stack-based algorithm with input the example in figure 1.1. Integers to the left of the edges are their insertion index. "Delete 3. (1,5)" in (b) means that edge $(1,5)$, inserted with index 3, is deleted. Similarly for the others. This figure is explained in the text above it.

These ideas are laid out in pseudocode 2.7. We include it here because it provides a first approximation to a more efficient implementation, given in pseudocode 2.9 with plenty of details, and because it is easier to reason about this algorithm's correctness with it.

Algorithm 2.7: Stack-based algorithm to calculate $C_G(\pi)$.

Input: $G = (V, E)$ a graph, π a linear arrangement of the vertices.
Output: $C_G(\pi)$, the total number of crossings.

```
1  Function CROSSINGS-STACKBASED(G, π) is
2      C ← 0, the number of crossings
3      S ← ∅, empty stack
4      for u ∈ V do
5          Sort Γ_u with respect to edge length in π.
6          Split Γ_u into Γ_u^+ and Γ_u^−.
7      for i from 1 to n do
8          u = π^{-1}(i)
9          for e ∈ Γ_u^+ in increasing edge length do
10             Find e in the stack S.
11             C ← C+ the amount of edges on top of e in S
12             Remove e from S.
13         for e ∈ Γ_u^− in decreasing edge length do
14             Push e into S.
15     return C
```

The index that we used in the example above is used in the data structure that actually implements the "stack" of the algorithm. The data structure used to implement the stack is a self-balancing binary search tree (BST) which stores pairs of integers and edges so that comparisons between edges (in order to decide how to store them inside the tree) can be done using this index. This tree is needed to locate quickly (in logarithmic time in the size of the tree) the edges inside the "stack" so as to delete them and, while doing so, to count how many edges are "on top of it".

Now we explain how to delete an edge and count, at the same time, how many elements are on top of it in the stack. First, notice that for a given pair $\langle i, e\rangle$ in the stack the pairs "on top" of it are those $\langle i', e'\rangle$ with $i' > i$. In other words, these indices always decrease from the top of the stack to its bottom. Therefore, counting the amount of such $\langle i', e'\rangle$ is fairly easy and can be done during the removal operation in line 12 of algorithm in pseudocode 2.7 with very little overhead added to the deletion operation. We detail this operation in pseudocode 2.8. In words, this operation looks for a pair $\langle i, e\rangle$ in the tree starting at its root and recursively looks for it in its subtrees as usual. Namely, if the search has reached node v which contains a pair $\langle i_v, e_v\rangle \neq \langle i, e\rangle$ then the search recursively looks for $\langle i, e\rangle$ in the left or right subtree according to the value of i_r. If $i_v > i$ then it branches to the left subtree (line 5), and when $i_v < i$ it branches to the left (line 9). Now, notice that when branching to the left subtree the pairs $\langle i', e'\rangle$ in the right subtree are on top of $\langle i, e\rangle$ in the stack since, by construction, $i' > i_v > i$. These pairs, and the pair in node v, need to be counted. This is done in line 6. When the search finds a node v that contains $\langle i, e\rangle$ (line 2) we count how many pairs are in its right subtree (again, by construction these are the pairs $\langle i', e'\rangle$ with $i' > i'$) and delete node v as it is usually done in self-balancing BSTs.

Algorithm 2.8: Deleting a pair of index and edge from the tree.

Input: S a self-balancing BST. One of its nodes v. $\langle i, e\rangle$ a pair of index and edge which is assumed to be in S.
Output: The amount of pairs $\langle i', e'\rangle$ such that $i' > i$. The pair $\langle i, e\rangle$ is removed from the tree.

```
1  Function RemoveEdgeFromTree_Rec(S, v, ⟨i, e⟩) is
2      if index_edge(v) = i then
           // The pair ⟨i, e⟩ has been found at node v of the tree.
           // Retrieve the size of the right subtree.
3          t ← right_child(v).size()
           // Remove pair from the tree.  At this step the tree must be rebalanced.
4          return t
5      else if index_edge(v) > i then
           // Retrieve the size of the right subtree.
6          t ← right_child(v).size() + 1
           // Remove pair from the left subtree, and retrieve the amount of pairs
              with index i' > i in that subtree.
7          t' ← RemoveEdgeFromTree_Rec(S, left_child(v), ⟨i, e⟩)
8          return t + t'
9      else
           // Remove pair from the right subtree, and retrieve the amount of pairs
              with index i' > i in that subtree.
10         t' ← RemoveEdgeFromTree_Rec(S, right_child(v), ⟨i, e⟩)
11         return t'
```

Input: S a self-balancing BST. $\langle i, e\rangle$ a pair of index and edge which is assumed to be in S.
Output: The amount of pairs $\langle i', e'\rangle$ such that $i' > i$. The pair $\langle i, e\rangle$ is removed from the tree.

```
12 Function RemoveEdgeFromTree(S, ⟨i, e⟩) is
13     return RemoveEdgeFromTree_Rec(S, root(S), ⟨i, e⟩)
```

Interestingly enough, there is yet another improvement that can be made in the algorithm. We do not actually need to push the edges in Γ_u^- into the "stack" one by one (line 13 of algorithm 2.7). It is easy to see that when we are processing vertex u all edges in Γ_u^- have larger insertion index than the current elements in the tree, and they are in sorted, also by insertion index. Therefore, we can construct a balanced BST of the edges in Γ_u^- in time $O(|\Gamma_u^-|)$ and perform a union operation of the resulting balanced BST and S. There is a straightforward algorithm that performs this operation in time $O(|t_1| + |t_2|)$ where $|t_1|$ and $|t_2|$ are the sizes of the trees being joined. However, the union of two AVL trees [1] can be performed in time $O(\log(|t_1| + |t_2|))$. The choice of AVL trees to implement the so-called stack is, therefore, convenient.

The algorithm that uses the tree is detailed in pseudocode 2.9. It starts by splitting every Γ_u into Γ_u^+ and Γ_u^- (line 5), with the particularity that Γ_u^- contains pairs of index and edge (initialised in line 4) and edges must be sorted in decreasing edge length. Then it computes every edge's index in loop of line 10. This can be done since we know in which order we push them into the stack. Notice that the edges in Γ_u^- must be assigned their corresponding index. The number of crossings is computed in loop of line 16. The union operation of two AVL trees is done in line 20 where it is assumed that Γ_u^- contains pairs of index and edge and that it is sorted (by index, or, equivalently, in decreasing edge length). Recall that the tree (line 2) has to contain pairs of insertion index and edge and store these pairs sorted by the value of the insertion index. Line 19 uses algorithm 2.8 to delete an edge from the tree and count, at the same time, how many edges have a larger insertion index.

In proposition 2.6 we prove this algorithm's correctness and give its time and space complexities.

Algorithm 2.9: Stack-based algorithm to compute $C_G(\pi)$ in time $O(m \log k_{max})$, space $O(m)$.

Input: $G = (V, E)$ a graph, π a linear arrangement of the vertices.
Output: $C_G(\pi)$, the total number of crossings.

```
1  Function CrossingsStackBased(G, π) is
2      S ← ∅, empty self-balancing BST
3      Γ_u^+ ← ∅, incoming edges ∀u ∈ V
4      Γ_u^- ← ∅, pairs of index and outgoing edges ∀u ∈ V
5      for u ∈ V do
6          Sort Γ_u with respect to edge length in π.
7          Split Γ_u into Γ_u^+ and Γ_u^-.
           // The pairs in Γ_u^- must be sorted in decreasing edge length in π.
       // Assign to every edge in Γ_u^- its corresponding insertion index.
8      H ← ∅, empty hash table relating edges and insertion index
9      I ← 0 insertion index
10     for i from 1 to n do
11         u = π^{-1}(i)
12         for e ∈ Γ_u^- do
13             I ← I + 1
14             H[e] ← I
               // Assign edge e ∈ Γ_u^- the index I inside Γ_u^-.
       // Count the edge crossings C_G(π).
15     C ← 0, the number of crossings
16     for i from 1 to n do
17         u = π^{-1}(i)
18         for e ∈ Γ_u^+ in increasing edge length do
               // Use algorithm in pseudocode 2.8.
19             C ← C + RemoveEdgeFromTree(S, ⟨H[e], e⟩)
           // The union between S and the BST containing the elements in Γ_u^-.
20         S ← S ∪ Γ_u^-
21     return C
```

Proposition 2.6. *Given a graph $G = (V, E)$ and a linear arrangement π, algorithm 2.9 computes $C_G(\pi)$. It does so in time $O(m \log k_{max})$ and space $O(m)$.*

Proof of correctness. We first prove its correctness. Although this proof is for algorithm in pseudocode 2.9, for this part of the proof we refer to algorithm in pseudocode 2.7 since these two are equivalent, namely, the result produced by algorithm 2.7 is a natural number c_1 if, and only if the result produced by algorithm 2.9 is $c_2 = c_1$. This can be easily seen by noticing that all the amount of edge pairs reported in 2.7 are also reported in algorithm 2.9 using the tree data structure. The proof of correctness is aimed at showing that $c_1 = C_G(\pi)$ and is based on previous work by Kosmas Palios and Georgios Pitsiladis.

The proof of correctness is given in three steps. In the first step we formalise all the invariants the stack is subject to. In the second we show that just right before removing an edge e from the stack this edge crosses all the edges on top of it. In the third, and last, step we show that the value returned by the algorithm is exactly the number of crossings. In the arguments to follow we assume that all edges are represented as ordered pairs (s, t) such that $\pi(s) < \pi(t)$.

First step: invariants The stack in algorithm 2.7 is subject to the following three invariants.

1. Let $e_1 = (s, t)$ be a fixed edge in the stack.
   (a) Any edge $e_2 = (u, v)$ above e_1 in the stack has $\pi(s) \leq \pi(u)$. That is true because e_2 was inserted after e_1 and edges are processed in the order of their leftmost vertices in π (line 7).
   (b) The edges $e_3 = (s, w)$ above e_1 are shorter since they are inserted in decreasing edge length, i.e., $\pi(w) < \pi(t)$ (line 13), and so are the edges $e_4 = (x, t)$ above e_1 because they are inserted in the order they appear in π, i.e. $\pi(s) < \pi(x)$.
2. An immediate consequence of invariant 1b is that at any step i of the algorithm, edge $e_j \in \Gamma_t^+ = (e_1, e_2, \cdots, e_p)$, $p = |\Gamma_t^+|$ and $t = \pi^{-1}(i)$, is below the edge e_{j-1} and above e_{j+1}.
3. Assume the algorithm has executed $i-1$ iterations, namely it has processed the first $i-1$ vertices.
   (a) All edges $e = (l, r)$ in the stack are such that $i \leq \pi(r)$, because they have been removed from the stack (within the loop in line 9). In other words, all edges with $\pi(r) < i$ are removed from the stack by the time the algorithm has reached iteration i.
   (b) Moreover, all edges $e = (l, r)$ have $\pi(l) < i$, because edges with leftmost vertex the vertex $\pi^{-1}(i)$ have not been added yet.

Edge crossings Assume that the algorithm has reached step i. Let $t = \pi^{-1}(i)$. In line 11 are counted all the edges that cross with the edge $e = (s, t) \in \Gamma_t^+$. In this step we assume e_1 to be the edge in the highest position in the stack, i.e. we assume e_1 to be the shortest edge with rightmost vertex t, the first found in Γ_t^+. Let $(l_1, r_1), (l_2, r_2), \cdots, (l_q, r_q)$ be the edges on top of e_1 before it is removed in line 12. Now we show that all (l_j, r_j) cross with edge e_1.

Firstly, due to invariant 3b,

$$\pi(l_j) < i = \pi(t), \quad \forall j \; 1 \leq j \leq q. \tag{2.6}$$

Secondly, invariant 1a tells us that $\pi(s) \leq \pi(l_j)$. In fact, we can show that the inequality is strict with the help of invariant 1b. The inequality is strict if, and only if, there does not exist an edge with $l_j = s$ above e_1. That is the case because all edges (s, r_j) above e_1 are shorter than e_1, namely, $\pi(r_j) < \pi(t)$, and thus have been deleted from the stack in a previous iteration $i' < i$. Therefore,

$$\pi(s) < \pi(l_j), \quad \forall j \; 1 \leq j \leq q. \tag{2.7}$$

Finally, invariant 3a tells us that $i \leq \pi(r_j)$. Again, the inequality is strict, and we use invariant 1b to show it. If an edge (l_j, t) existed then it would be shorter than e_1, but it can not happen due to our assumption that e_1 is the shortest edge with rightmost vertex t. Therefore,

$$\pi(t) < \pi(r_j), \quad \forall j \; 1 \leq j \leq q. \tag{2.8}$$

Putting together these results (2.6, 2.7 and 2.8) we obtain the definition of crossing in equation 1.2,

$$\pi(s) < \pi(l_j) < \pi(t) < \pi(r_j) \tag{2.9}$$

In short, all edges (l_j, r_j) above the first edge $e_1 \in \Gamma_t^+$ in the stack cross edge e_1 which, after accumulating the amount of edges above it, it is deleted. Therefore, since the edges in Γ_t^+ are processed in a top-down fashion, namely in increasing edge length (invariant 2) in the stack, when processing any edge $e_{j'} \in \Gamma_t^+$ the other edges $e_{j''} \in \Gamma_t^+$ with $j'' < j'$ are no longer in the stack, and the same edges (l_j, r_j), now looked from $e_{j'}$, are also above $e_{j'}$ and also cross with e_q because of the same reasons explained above.

Exactly $C_G(\pi)$ crossings are counted The fact that every edge crossing is counted can be easily proved. Take any two edges, say $e_1 = (s, t)$ and $e_2 = (u, v)$, with $t = \pi(i)$. If e_1 and e_2 cross we may assume, without loss of generality, that $\pi(s) < \pi(u) < i < \pi(v)$. Due to the insertion order of the edges in the algorithm, e_2 is above e_1 on the stack ($\pi(s) < \pi(u)$). At the i-th step, just before e_1 is deleted, e_2 is still in the stack and above e_1 ($i < \pi(v)$).

In case e_1 and e_2 do not cross, e_2 could be, at some point, above e_1, but not just right before deleting e_1. This is due to the fact that if edges do not cross we must have one of

$$\pi(u) < \pi(v) < \pi(s) < i \tag{2.10}$$
$$\pi(s) < \pi(u) < \pi(v) < i \tag{2.11}$$
$$\pi(s) < i < \pi(u) < \pi(v) \tag{2.12}$$

When the algorithm has reached step i, just right before deleting e_1, in cases 2.10 and 2.11 e_2 is not in the stack because it was deleted in a previous step $i' < i$ (when processing vertex v, i.e., $i' = \pi(v)$), in case 2.12 the edge e_2 is not in the stack because it still has to be added in a future step $i < i' = \pi(u)$. □

Proof of complexity. here we analyse the actual stack-based algorithm, given in pseudocode 2.9 . The space that the tree can take up is at most $O(m)$. There are two worst cases: that of a linear arrangement of the vertices of a star tree where the hub is placed at one of the ends of the linear arrangement, or that of a complete graph where, at the end of iteration $n/2$, the tree has size

$$\sum_{i=1}^{n/2}(|\Gamma_i^-| - |\Gamma_i^+|) = \sum_{i=1}^{n/2}((n-i)-(i-1)) = \sum_{i=1}^{n/2}(n-2i+1) = \frac{n^2}{4} = O(m).$$

Moreover, while the size of the stack is at most $O(m)$, the size of the hash table H (line 8) is exactly m, since it has to store an index for each edge.

The time complexity is also immediate. The sorting stage takes (loop in line 5) $O(m \log k_{max})$ time, since

$$\sum_{i=1}^{n} k_i \log k_i \leq \sum_{i=1}^{n} k_i \log k_{max} = 2m \log k_{max}.$$

The second step (assigning an index to every edge, line 10) takes $O(m)$ time, assuming a constant-time cost of inserting each edge into the table. The third and last step, i.e., the computation of $C_G(\pi)$ (line 16) has cost, for a single vertex u,

$$O\left(k_u \log |S_i| + \log\left(|S_i| - |\Gamma_u^+|\right)\right).$$

$|S_i|$ denotes the size of the tree at the beginning of iteration i. Since the size of the tree can be at most $O(m)$, as shown before, then there is some iteration $1 \leq i \leq n$ for which the cost is

$$k_u \log |S_i| + \log\left(|S_i| - |\Gamma_u^+|\right) \leq k_u \log m + \log m = O(k_{max} \log m).$$

Therefore, even though the step that actually computes the amount of crossings is quite efficient, the steps that are the heaviest are the sorting of the edges and the obtention of the insertion indices. □

Needless to say that, when G is a tree the time and space complexities change to $O(n \log n)$ and $O(n)$, respectively.

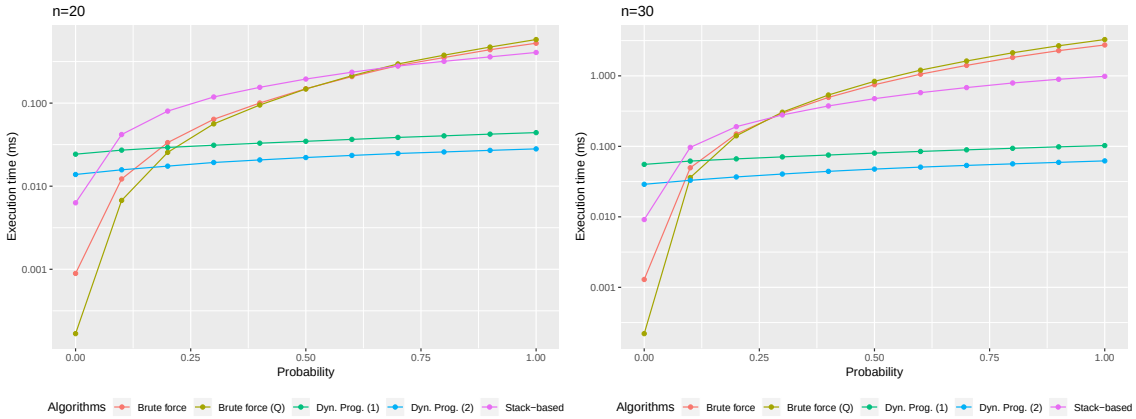
2.4 Performance comparison

We devote this section to comparing the performance of each algorithm. In order to do so, we designed two very simple experiments. In one we use random graphs $G_{n,p}$, and in the other we use labelled trees

generated uniformly at random using Prüfer codes [27]. In both of them we measured the execution time of our C++ implementations, without any optimisation of any kind on the compiler's side, of the algorithms detailed above. Besides, we also measured the execution time of the algorithms briefly described at the beginning of this section (see pseudocodes 2.1 and 2.2) for comparison purposes. Notice that algorithm 2.1 needs to know the elements of Q. In a practical application, since $|Q|$ is large, the algorithm would try to enumerate its elements and while doing so compute the number of crossings. However, for this algorithm, we decided to split this process into enumeration and computation of the number of crossings. The execution of the first part was not measured. In the following figures we present the average execution time to compute $C_G(\pi)$ for a single linear arrangement for the dynamic programming algorithms (pseudocodes 2.5 and 2.6) and the stack-based algorithm 2.6.

Both experiments consist on measuring the time that every algorithm takes to compute the number of crossings in one linear arrangement of its vertices. For this, several linear arrangements were used in both experiments and were generated uniformly at random. In the first experiment, we generated 150 $G_{n,p}$ uniformly at random for each pair of values n and p. For any value of n, the probability parameter that two edges are connected, p, always takes its value from $\{0.0, 0.1, 0.2, \cdots, 1.0\}$. The amount of linear arrangements used was 10^3 for $n \leq 50$, and 100 for $100 \leq n \leq 10^3$. The second experiment is simpler. We generated all the unlabelled free trees, for $n = 2, 3, 4, \cdots, 15$, and generated trees uniformly at random for $n = 20, 30, 40, 50, 100, 250, 500, 1000$. The amount of linear arrangements is 10^4 for $n \leq 15$ and 1000 for $n \geq 20$.

The results of the first experiment are very clear. While it seems that the stack-based algorithm beats the brute force algorithms for modest values of n (see figure 2.7 for values $n = 20, 30, 40, 50$) there does not seem to be any reason to make us think that the same will happen with respect to the dynamic programming algorithms. The fastest algorithm, for denser graphs, seems to be the second dynamic programming algorithm, presented in section 2.2. This is confirmed in figures 2.7 of values $n = 500$ and $n = 1000$.



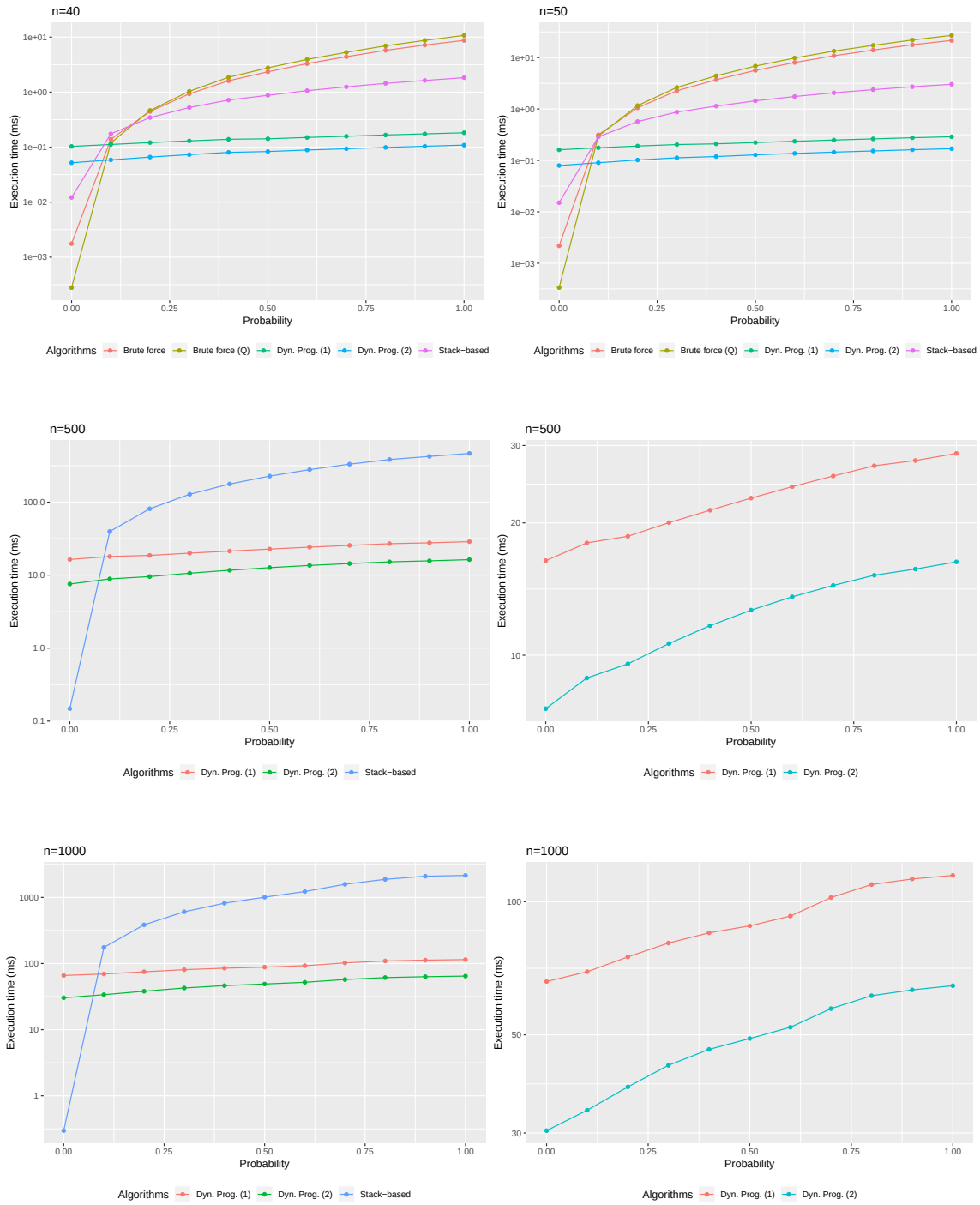


Figure 2.7: Execution times of the algorithms explained above for different values of n, all executed in several $G_{n,p}$. Algorithms: Brute force (Q) 2.1, Brute force 2.2, Dyn. Prog (1) 2.5, Dyn. Prog. (2) 2.6, and Stack-based 2.9.

Moreover, besides showing that the time complexities of the dynamic programming algorithms are the same, i.e., $O\left(n^2\right)$ as shown in propositions 2.4 and 2.5, the stack-based algorithm distances itself from the these algorithms in a non-linear fashion.

The second experiment, however, shows that the stack-based algorithm eventually becomes faster than the two dynamic programming algorithms. We have been able to collect enough data to show that for $n = 1000$ the stack-based algorithm is faster, but the smallest value of n for which this happens is probably quite smaller. Furthermore, It confirms that the stack-based algorithm has a lower asymptotic cost when executed on trees. This can be seen in figure 2.8.

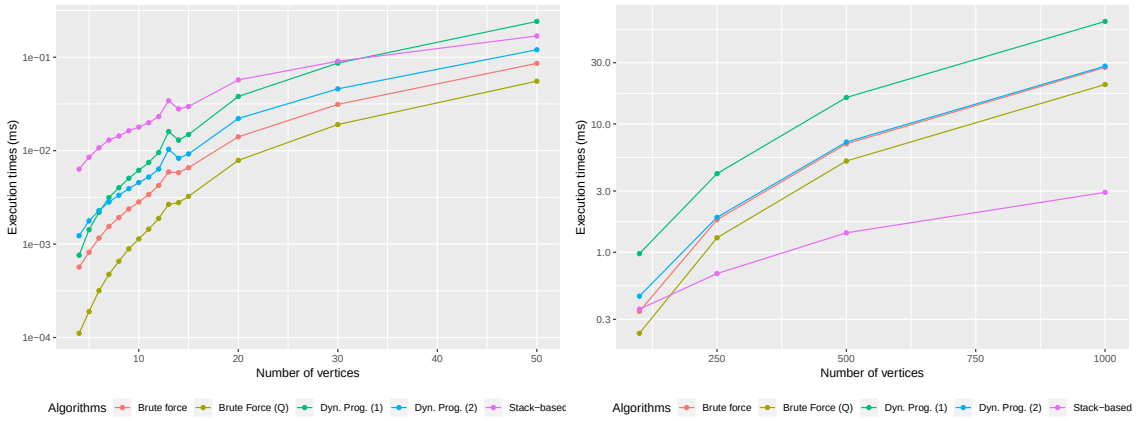


Figure 2.8: Execution times of the algorithms explained above for different values of n, all executed in random labelled trees of size n. Algorithms: Brute force (Q) 2.1, Brute force 2.2, Dyn. Prog (1) 2.5, Dyn. Prog. (2) 2.6, and Stack-based 2.9.

It is worth mentioning that the results presented in figure 2.8 (and also in figure 2.7) are the execution times of a non-optimised compilation of the C++ implementations. Notice, for instance, that the execution time for the brute force algorithm that uses Q is faster than the second dynamic programming algorithm as shown in figure 2.8. When the implementations are compiled under strong optimisation flags (e.g., `-O3` for the `g++` compiler) the execution times of the dynamic programming algorithms plummet, whereas those of the brute force algorithms do not. Therefore, the conclusions of the experiments change under optimisation in that the brute force algorithms are actually slower than the dynamic programming ones. However, the first dynamic programming algorithm is still slower than the second, and the stack-based algorithm remains the fastest. This is shown in figure 2.9. Interestingly enough, figures 2.8 and 2.9 hint that, when restricted to trees, the brute force algorithms might have the same asymptotic cost as that of the dynamic programming algorithms. This does not apply to the stack-based.

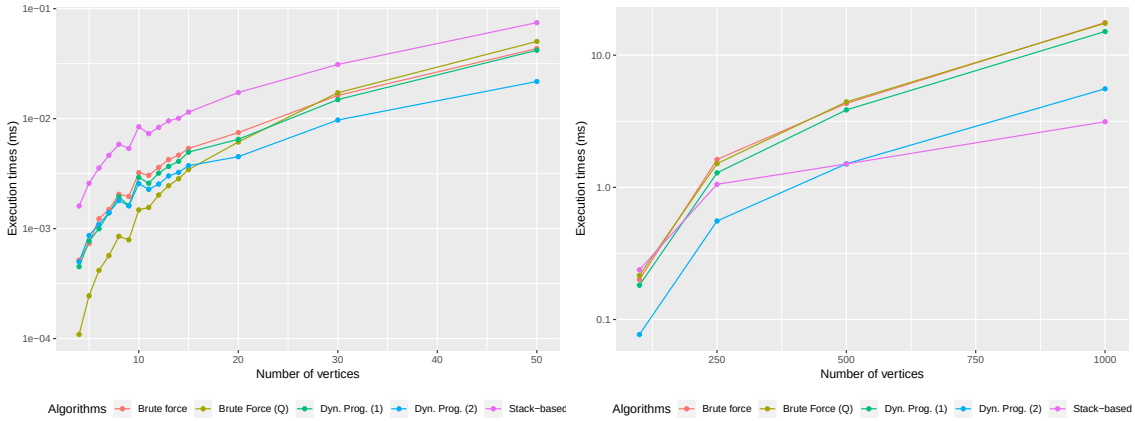


Figure 2.9: Execution times of the optimised implementations (using `g++` and optimisation flags `-O3` and `-DNDEBUG`) of the algorithms explained above for different values of n, all executed in random labelled trees of size n. Algorithms: Brute force (Q) 2.1, Brute force 2.2, Dyn. Prog (1) 2.5, Dyn. Prog. (2) 2.6, and Stack-based 2.9.

3 Expectation of the number of crossings

In this section we investigate a little bit further the expectation of the number of crossings C_G in a graph G. In section 3.1 we give a mathematical expression for this expectation in random graphs from $\mathcal{G}_{n,p}$. In section 3.2 we revisit an existing work by Ferrer-i-Cancho (see [6]) where the expectation of the number of crossings was studied, not on the space of uniformly random linear arrangements, but on the space of those linear arrangements for which the length of each edge is given by a certain function.

3.1 In random graphs

In this section we study the expected amount of crossings over the space of random graphs $\mathcal{G}_{n,p}$, and over the space of uniformly random linear arrangements, namely $\mathbb{E}_{n,p}\left[\,\mathbb{E}_{rla}\left[\,C_{G_{n,p}}\,\right]\,\right]$. This is given in the form of a single proposition for which we provide two proofs. The first, longer than the second, proves it by means of simple algebraic calculations. The second, proves it via a very interesting result by Bollobás.

The result obtained was validated by estimating the average amount of crossings in several graphs $G_{n,p} \in \mathcal{G}_{n,p}$. This is depicted in figure 3.1 which shows the value of equation 3.1, which gives the expected amount of crossings in $G_{n,p}$, for several values of n and p.

Proposition 3.1. *The expected amount of crossings of any graph $G_{n,p} \in \mathcal{G}_{n,p}$ in a uniformly random linear arrangement is*

$$\mathbb{E}_{n,p}\left[\,\mathbb{E}_{rla}\left[\,C_{G_{n,p}}\,\right]\,\right] = \frac{1}{3}\,\mathbb{E}_{n,p}\left[\,X_Q\,\right] = \binom{n}{4}p^2. \tag{3.1}$$

Long proof. In this proof, we derive equation 3.1 through an analysis of the expected size of the set Q in random graphs. Notice that the first equality follows directly from equation 1.4.

Let $X_Q = |Q(G_{n,p})|$ be a random variable. Its expectation, over the space of random graphs on n and p, $\mathbb{E}_{n,p}\left[\,X_Q\,\right]$, is given in equation 3.2. In order to compute it, we use equation 1.5 to get

$$\begin{aligned}\mathbb{E}_{n,p}\left[\,X_Q\,\right] &= \mathbb{E}_{n,p}\left[\,\frac{1}{2}\left(X_m^2 + X_m - \sum_{u\in V(G_{n,p})} D_u^2\right)\,\right] \\ &= \frac{1}{2}\left(\mathbb{E}_{n,p}\left[\,X_m^2\,\right] + \mathbb{E}_{n,p}\left[\,X_m\,\right] - \sum_{u\in V(G_{n,p})} \mathbb{E}_{n,p}\left[\,D_u^2\,\right]\right).\end{aligned}$$

where $X_m = |E(G_{n,p})|$ and $D_u = k_u$ are integer-valued random variables. It is well-know that $X_m \sim B(\binom{n}{2}, p)$ and $D_u \sim B(n-1, p)$, where $B(k, q)$ denotes a binomial distribution with amount of trials k and probability of success q. Therefore

$$\mathbb{E}_{n,p}\left[\,X_m^2\,\right] = \mathbb{V}_{n,p}\left[\,X_m\,\right] + \mathbb{E}_{n,p}\left[\,X_m\,\right]^2 = \frac{1}{4}pn(n-1)(2 + p(n-2)(n+1)),$$

and

$$\mathbb{E}_{n,p}\left[\,D_u^2\,\right] = \mathbb{V}_{n,p}\left[\,D_u\,\right] + \mathbb{E}_{n,p}\left[\,D_u\,\right]^2 = (n-1)p(1 + p(n-2)).$$

By plugging these results into the first equation we get, after some algebra

$$\mathbb{E}_{n,p}\left[\,X_Q\,\right] = 3\binom{n}{4}p^2. \tag{3.2}$$

This completes the longer proof. □

Short proof. The same result can be achieved by making use of equation [5, section VII]

$$\mathbb{E}_{n,p}[\,X_F\,] = n_{\mathcal{K}_n}(F)p^{|E(F)|}$$

where $X_F = n_{G_{n,p}}(F)$ is the number of subgraphs of $G_{n,p}$ isomorphic to F. We know that, as pointed out in [2], for any graph G we have

$$|Q(G)| = n_G(\mathcal{L}_2 \oplus \mathcal{L}_2).$$

Therefore we can conclude that

$$\mathbb{E}_{n,p}[\,X_{\mathcal{L}_2\oplus\mathcal{L}_2}\,] = n_{\mathcal{K}_n}(\mathcal{L}_2 \oplus \mathcal{L}_2)p^{|E(\mathcal{L}_2\oplus\mathcal{L}_2)|} = 3\binom{n}{4}p^2. \tag{3.3}$$

□

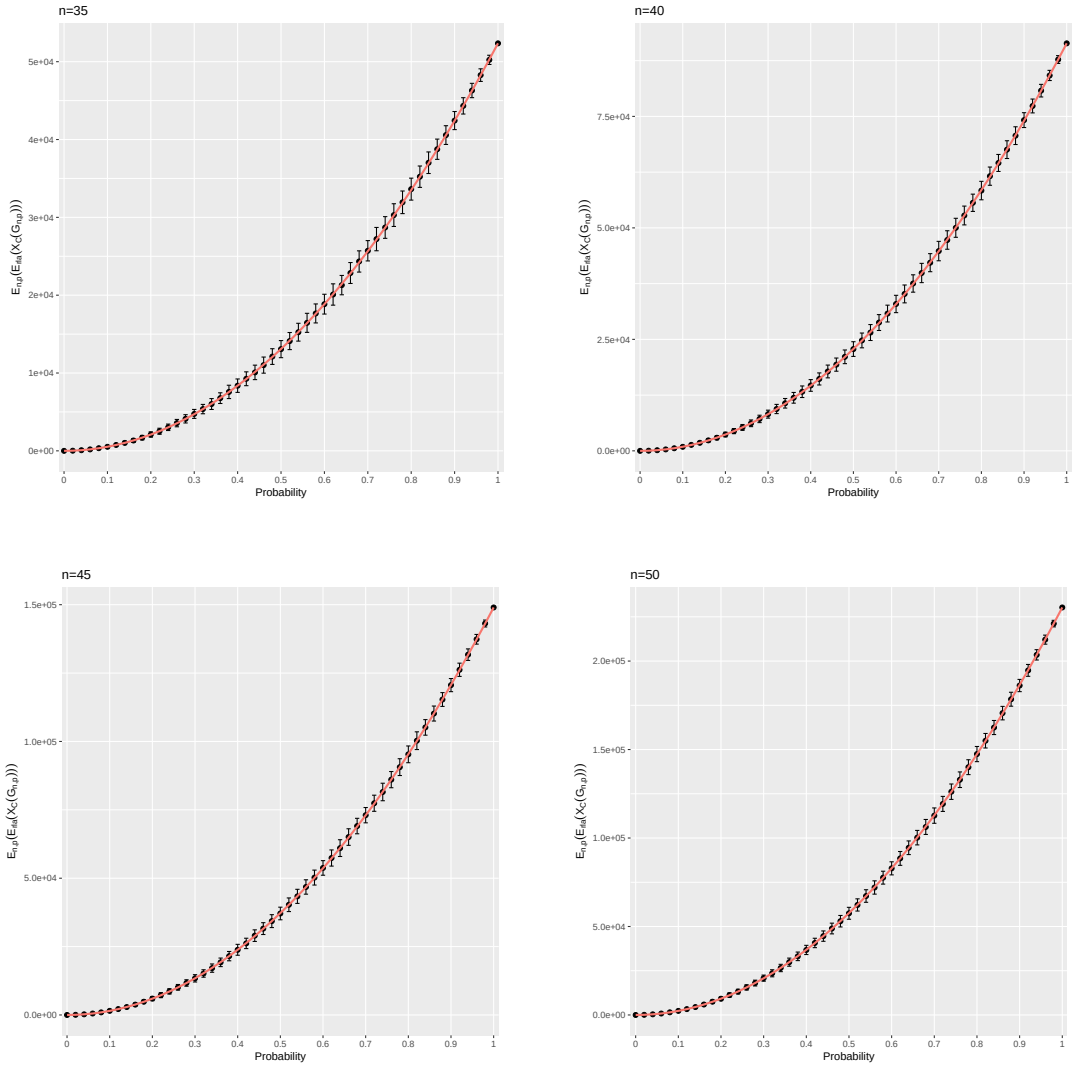


Figure 3.1: Value of equation 3.1 (red line) for different values of n and p. For each value of n, we generated uniformly at random 10000 linear arrangements. For a fixed value of n we generated 500 random graphs for each value of p and plotted with black dots the average of the amount of crossings for all linear arrangements.

3.2 As a function of the lengths of the edges

In previous work (see [6]), Ferrer-i-Cancho presented a better approximation to the real value of C in syntactic dependency trees than $\mathbb{E}_{rla}[C_G]$, first presented here in equation 1.4. In his work, he aimed at giving the expected amount of crossings given the length of the edges. This expectation, as opposed to the first defined in equation 1.4, is defined over the set of linear arrangements that have the same edge length, for each edge of the graph. Recall equation 1.6 where the function θ_π defines the length of an edge in a linear arrangement. We define the set of all linear arrangements of the vertices of a graph that yield the same edge lengths as specified by a certain fixed function θ_0

$$\Pi(\theta_0) = \{\pi \mid \forall e \in E,\ \theta_\pi(e) = \theta_0(e)\}.$$

For example, in $\mathcal{C}_4$ vertices we can define a length function θ_0 such that $\theta_0(\{1,2\}) = 2$, $\theta_0(\{2,3\}) = 1$, $\theta_0(\{3,4\}) = 2$ and $\theta_0(\{1,4\}) = 3$. Then, the linear arrangements that make the edges of $\mathcal{C}_4$ have the lengths specified by θ_0 are $\Pi(\theta_0) = \{(1,3,2,4),(4,2,3,1)\}$. The expectation of C_G when given full knowledge of the edges, determined by a fixed function θ_0, can be defined as

$$\mathbb{E}_{\Pi(\theta_0)}[C_G] = \sum_{\{st,uv\}\in Q} \Pr[c(st,uv) = 1 \mid \theta_0]. \tag{3.4}$$

According to Ferrer-i-Cancho, this expectation makes a better prediction than equation 1.4 of the number of crossings. Although the exact complexity of evaluating equation 3.4 was not studied, a direct implementation is computationally expensive due to the necessity of finding all the linear arrangements in $\Pi(\theta_0)$. For this reason, and the need of designing a parsimonious predictor he gave another expression that is simpler to calculate, approximates well enough equation 3.4 and is still better than equation 1.4. This new expression is

$$\mathcal{X}(\pi) = \sum_{\{st,uv\}\in Q} \Pr[\mathit{crossing} \mid \theta_\pi(st), \theta_\pi(uv)] \tag{3.5}$$

In this new expression, we require the probability that two independent edges of lengths d_1 and d_2 cross when arranged at random on a linear arrangement. This probability is

$$\Pr[\mathit{crossing} \mid d_1, d_2] = \frac{|\alpha(d_1,d_2)|}{|\beta(d_1,d_2)|} \tag{3.6}$$

where $\alpha(d_1,d_2)$ and $\beta(d_1,d_2)$ are sets containing pairs of integers s_1 and s_2, each representing the starting position of an edge in the linear arrangement (the left-most vertex), whose lengths are d_1 and d_2 respectively. The difference between the two sets is that $\alpha(d_1,d_2)$ contains the locations of the edges that cross, and $\beta(d_1,d_2)$ contains those that cross and that do not cross.

The formal definitions of $\alpha(d_1,d_2)$ and $\beta(d_1,d_2)$ are simple. First, a pair of valid starting positions of two edges s_1 and s_2, of length d_1 and d_2 respectively, are those pairs (s_1,s_2) such that

$$\{s_1, s_1+d_1\} \cap \{s_2, s_2+d_2\} = \emptyset, \quad s_1 \le n-d_1, s_2 \le n-d_2.$$

With this, we can define $\alpha(d_1,d_2)$ and $\beta(d_1,d_2)$

$$\begin{aligned}\alpha(d_1,d_2) = \{(s_1,s_2) \mid\ & s_1 \text{ and } s_2 \text{ are valid positions} \wedge \\ & (s_1 < s_2 < s_1+d_1 < s_2+d_2 \vee \\ & s_2 < s_1 < s_2+d_2 < s_1+d_1)\},\end{aligned} \tag{3.7}$$

$$\beta(d_1,d_2) = \{(s_1,s_2) \mid s_1 \text{ and } s_2 \text{ are valid positions}\}. \tag{3.8}$$

Figure 3.2 illustrates the values of equation 3.6 for several pairs of lengths $1 \le d_1, d_2 \le n$, for $n \in \{4,8,16,32\}$.

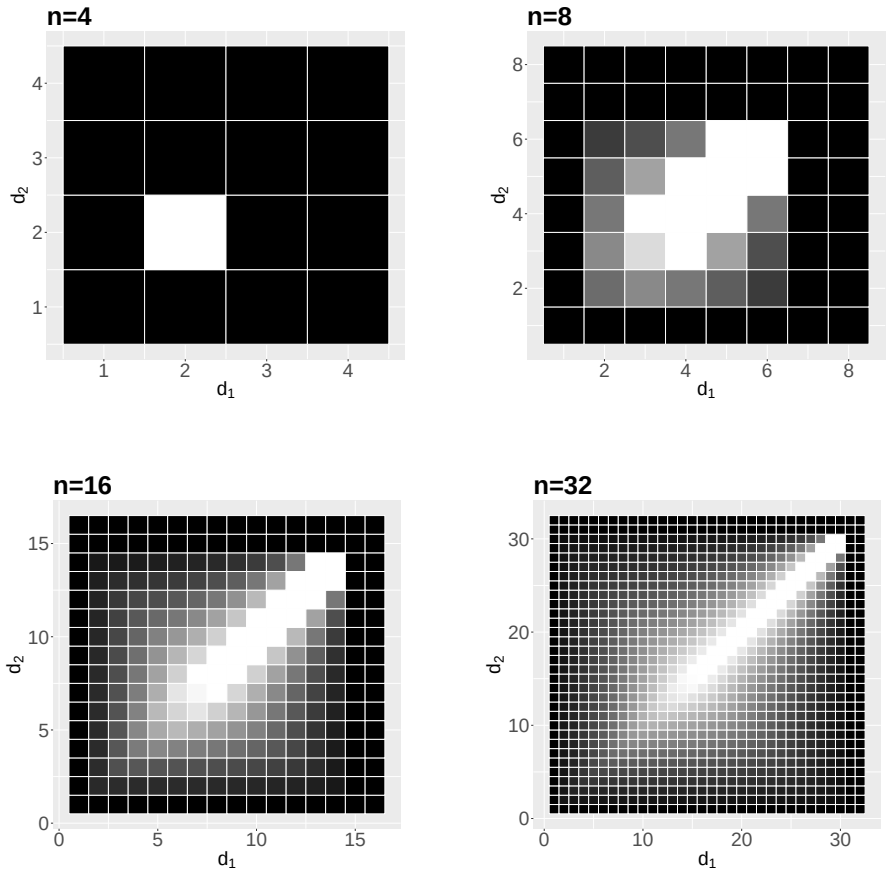


Figure 3.2: Heat maps of the values $|\alpha(d_1, d_2)|/|\beta(d_1, d_2)|$ for all possible pairs of $1 \leq d_1, d_2 \leq n$. The value of n is indicated at the top left corner of each heat map.

The sizes of $\alpha(d_1, d_2)$ and $\beta(d_1, d_2)$ can be calculated by means of brute force algorithms, presented in pseudocodes 3.1 and 3.2. They are given so as to clarify any doubts on their definition. In both algorithms, and in the rest of algorithms to come in this section, we assume that n is an implicit parameter.

Algorithm 3.1: Computing $|\alpha(d_1, d_2)|$.

Input: n arrangement's size, d_1 and d_2 lengths of edges, $d_1 \leq d_2$.
Output: The size of $\alpha(d_1, d_2)$.

```
1 Function |α(d1, d2)| is
2     α ← 0
3     for s1 from 1 to n − d1 do
4         for s2 from 1 to n − d2 do
5             if {s1, s1 + d1} ∩ {s2, s2 + d2} ≠ ∅ then
                  // The edges have vertices in common.  The pair is not valid.
6                 Do nothing
7             else if s1 < s2 < s1 + d1 < s2 + d2 ∨ s2 < s1 < s2 + d2 < s1 + d1 then
                  // The edges cross.  The pair is valid.
8                 α ← α + 1
9     return α
```

Algorithm 3.2: Computing $|\beta(d_1, d_2)|$.

Input: n arrangement's size, d_1 and d_2 lengths of edges, $d_1 \leq d_2$.
Output: The size of $\beta(d_1, d_2)$.

```
1 Function |β(d1, d2)| is
2     β ← 0
3     for s1 from 1 to n − d1 do
4         for s2 from 1 to n − d2 do
5             if {s1, s1 + d1} ∩ {s2, s2 + d2} ≠ ∅ then
                  // The edges have vertices in common.  The pair is not valid.
6                 Do nothing
7             else β ← β + 1
8     return β
```

In the following subsections we provide algorithms to perform the same task in constant time. Since we are dealing with independent edges we should have $n \geq 4$ but the algorithms derived work for all $n \geq 1$. We omit the unnecessary details of some derivations because they follow from straightforward calculations. The results presented have been validated using the brute force algorithms above. The protocol of validation is detailed in section A.2.

In order to derive more easily an algorithm to compute the size of sets $|\alpha(d_1, d_2)|$ and $|\beta(d_1, d_2)|$, we split the problem into two parts. We first calculate the amount of pairs (s_1, s_2) within α and β such that $s_1 < s_2$ denoted as $|\alpha^+(d_1, d_2)|$ and $|\beta^+(d_1, d_2)|$. Then we calculate the amount of remaining pairs, those such that $s_1 > s_2$, denoted as $\alpha^-(d_1, d_2)$ and $\beta^-(d_1, d_2)$. Also, while deriving these algorithms, we assume, without loss of generality, that $d_1 \leq d_2$. Evidently,

$$|\alpha(d_1, d_2)| = \begin{cases} |\alpha^+(d_1, d_2)| + |\alpha^-(d_1, d_2)| & \text{if } d_1 \leq d_2, \\ |\alpha^+(d_2, d_1)| + |\alpha^-(d_2, d_1)| & \text{otherwise,} \end{cases} \tag{3.9}$$

and

$$|\beta(d_1, d_2)| = \begin{cases} |\beta^+(d_1, d_2)| + |\beta^-(d_1, d_2)| & \text{if } d_1 \leq d_2, \\ |\beta^+(d_2, d_1)| + |\beta^-(d_2, d_1)| & \text{otherwise.} \end{cases} \tag{3.10}$$

The reader should bear in mind that in order to have valid pairs we must have $s_1 \leq n - d_1$ and $s_2 \leq n - d_2$.

The derivations are explained with the help of figures with a common format. Pairs of cells crossed with a single line of the same colour are always at distance d_2. Cells crossed with multiple black lines are always at distance d_1.

3.2.1 $\alpha(d_1, d_2)$

We first give an algorithm to compute $|\alpha(d_1, d_2)|$ since we use these results to compute $|\beta(d_1, d_2)|$.

$s_1 < s_2$: In order to derive an expression for its size, we first compute the size of the subset of pairs (s_1, s_2) where s_1 is fixed, denoted as $\alpha^+(s_1; d_1, d_2)$. This is given in equation 3.11. Then we use it to derive the size of $\alpha^+(d_1, d_2)$ by summing over the appropriate values of s_1.

$$|\alpha^+_*(s_1; d_1, d_2)| = \begin{cases} d_1 - 1 & \text{if } s_1 + d_1 \leq n - d_2, \\ n - d_2 - s_1 & \text{if } n - (d_1 + d_2) < s_1 < n - d_2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.11}$$

The cases of equation 3.11 are indicated in figure 3.3. In 3.3(a) is shown the interval for the valid values of s_2 such that $s_1 < s_2 < s_1 + d_1 < n - d_2$, which correspond to the first case. In 3.3(b) is

shown the interval for the valid values of s_2 such that $s_1 < s_2 < n - d_2 < s_1 + d_1$, which correspond to the second case.

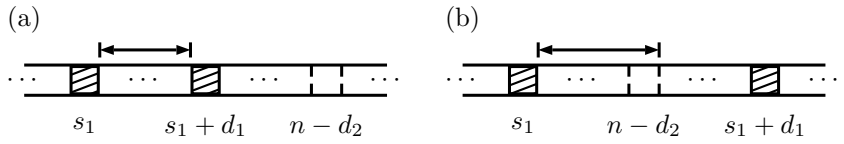


Figure 3.3: Given a fixed position s_1, the amount of valid positions s_2 such that $(s_1, s_2) \in \alpha_*^+(s_1; d_1, d_2)$ is, (a), $s_1 + d_1 - s_1 - 1$, and (b), $n - d_2 - s_1$.

The relationship between $n - d_2$ and the pair $(s_1, s_1 + d_1)$ affects how we compute, not only $|\alpha_*^+(s_1; d_1, d_2)|$, but also $|\alpha_*^-(s_1; d_1, d_2)|$, $|\beta_*^+(s_1; d_1, d_2)|$ and $|\beta_*^-(s_1; d_1, d_2)|$ in the coming derivations. For the sake of space we do not depict it in the figures. Notice $n - d_2$ is always between s_1 and $s_1 + d_1$, for any valid s_1 and $d_1 \leq d_2$. The equation $|\alpha^+(d_1, d_2)|$ is given in 3.12.

$$|\alpha^+(d_1, d_2)| = \begin{cases} (d_1 - 1)(n - d_1 - d_2) + d_1(d_1 - 1)/2 & \text{if } 1 \leq n - (d_1 + d_2), \\ (d_2 - n)(d_2 - n + 1) & \text{otherwise.} \end{cases} \tag{3.12}$$

The first case is obtained by

$$\sum_{s_1=1}^{n-(d_1+d_2)} (d_1 - 1) + \sum_{s_1=n-(d_1+d_2)+1}^{n-d_2-1} (n - s_1 - d_2) = (d_1 - 1)(n - d_1 - d_2) + d_1(d_1 - 1)/2,$$

and the second by

$$\sum_{s_1=1}^{n-d_2-1} (n - s_1 - d_2) = (d_2 - n)(d_2 - n + 1).$$

$s_1 > s_2$: Similarly, we first compute the size of the subset of pairs (s_1, s_2) where $s_1 \leq n - d_1$ is fixed, denoted as $\alpha^-(s_1; d_1, d_2)$. This is given in function 3.13.

$$|\alpha_*^-(s_1; d_1, d_2)| = \begin{cases} d_1 - 1 & \text{if } 1 \leq s_1 - d_2, \\ s_1 + d_1 - d_2 - 1 & \text{if } s_1 - d_2 < 1 \leq s_1 + d_1 - d_2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.13}$$

Recall that we assumed $d_1 \leq d_2$. The cases of equation 3.13 are indicated in figure 3.4.

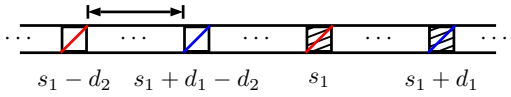


Figure 3.4: Given a fixed position s_1, the amount of valid positions s_2 such that $(s_1, s_2) \in \alpha_*^-(s_1; d_1, d_2)$ is indicated with an arrow.

This time we formalise the computation of $|\alpha^-(d_1, d_2)|$ with an algorithm, given in pseudocode 3.3. The justification of each case is given in the pseudocode in the form of comments.

Algorithm 3.3: Computing $|\alpha^-(d_1, d_2)|$.

Input: n arrangement's size, d_1 and d_2 lengths of edges, $d_1 \leq d_2$.
Output: The amount of pairs $(s_1, s_2) \in \alpha(d_1, d_2)$ such that $s_2 < s_1$.

Function $|\alpha^-(d_1, d_2)|$ **is**
  $\alpha^- \leftarrow 0$
  if $d_1 + d_2 \leq n$ **then**
    // Add the sum $\sum_{s_1=1+d_2}^{n-d_1}(d_1 - 1)$, first case of function 3.13
    $\alpha^- \leftarrow \alpha^- + (d_1 - 1)(n - d_1 - d_2)$
  if $1 + d_2 - d_1 \geq 1$ **then**
    // Second case of function 3.13 Add the sum $\sum_{s_1=1+d_2-d_1}^{x}(s_1 + d_1 - d_2 - 1)$.
    if $1 + d_2 \leq n - d_1$ **then**
      // Take $x = d_2$
      $\alpha^- \leftarrow \alpha^- + d_1(d_1 - 1)/2$
    else
      // Take $x = n - d_1$
      $\alpha^- \leftarrow \alpha^- + (n - d_2)(n - d_2 - 1)/2$
  return α^-

3.2.2 $\beta(d_1, d_2)$

Likewise, this is computed in two parts: we first count those pairs $(s_1, s_2) \in \beta(d_1, d_2)$ such that $s_1 < s_2$, denoted as $\beta^+(d_1, d_2)$, and then those $(s_1, s_2) \in \beta(d_1, d_2)$ such that $s_1 > s_2$, denoted as $\beta^-(d_1, d_2)$.

$s_1 < s_2$: By definition we have that $|\beta^+_*(s_1; d_1, d_2)| > |\alpha^+_*(s_1; d_1, d_2)|$. Equation 3.14 gives this size and like equation 3.11 it is also split into three cases. For the first case, we have to add to $|\alpha^+_*(s_1; d_1, d_2)|$ (recall equation 3.11) the pairs (s_1, s_2) such that $s_1 + d_1 < s_2 \leq n - d_2$. This can be seen with the help of figure 3.3(a). All the pairs in the second case are already contained in $\alpha^+_*(s_1; d_1, d_2)$ (figure 3.3(b)). Therefore

$$|\beta^+_*(s_1; d_1, d_2)| = |\alpha^+_*(s_1; d_1, d_2)| + \begin{cases} n - d_2 - (s_1 + d_1) & \text{if } s_1 + d_1 \leq n - d_2, \\ 0 & \text{if } n - (d_1 + d_2) < s_1 < n - d_2, \\ 0 & \text{otherwise} \end{cases}$$

$$= \begin{cases} n - s_1 - d_2 - 1 & \text{if } s_1 + d_1 \leq n - d_2, \\ n - s_1 - d_2 & \text{if } n - (d_1 + d_2) < s_1 < n - d_2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.14}$$

The procedure for computing the size of $\beta^+(d_1, d_2)$ is given in equation 3.15. The justification of each case is given in the pseudocode in the form of comments.

$$|\beta^+(d_1, d_2)| = \frac{1}{2}\begin{cases} (n - d_2)^2 + 3(d_1 + d_2 - n) - d_1^2 + d_1(d_1 - 1) & \text{if } 1 \leq n - (d_1 + d_2), \\ (d_2 - n)(d_2 - n + 1) & \text{otherwise.} \end{cases} \tag{3.15}$$

The first case is obtained by computing

$$\sum_{s_1=1}^{n-(d_1+d_2)} (n - s_1 - d_2 - 1) + \sum_{s_1=n-(d_1+d_2)+1}^{n-d_2-1} (n - s_1 - d_2),$$

and the second by

$$\sum_{s_1=1}^{n-d_2-1} (n - s_1 - d_2).$$

$s_1 > s_2$: We follow the same strategy, with some changes. This time we base the derivation on a similar figure to the one used to derive $|\alpha_*^-(s_1; d_1, d_2)|$, where we consider one more case for each of the two situations depicted in figure 3.4, namely $d_1 < d_2$. Since the resulting expression for the computation of $|\beta_*^-(s_1; d_1, d_2)|$ has too many cases to fit them inside an equation, we have encapsulated it in pseudocode 3.4.

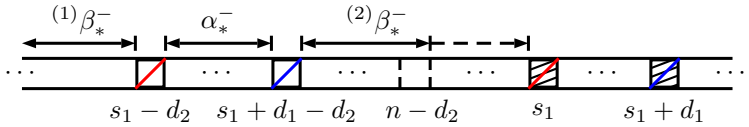


Figure 3.5: All cases to be taken considered to count the pairs in $\beta_*^-(s_1; d_1, d_2)$. This figure shows in total 3×2 cases to be considered to compute $|\beta_*^-(s_1; d_1, d_2)|$, for each of which we consider the binary case where $n - d_2 \leq s_1$ or $s_1 < n - d_2$. The first case is $1 \leq s_1 - d_2$, the second is $s_1 - d_2 < 1 \leq s_1 + d_1 - d_2$, and the third is $s_1 + d_1 - d_2 < 1$.

Figure 3.5 shows all the cases to consider while looking for an arithmetic expression for $|\beta_*^-(s_1; d_1, d_2)|$. When considering the first case, namely when $1 \leq s_1 - d_2$, we obtain a very simple procedure to derive $|\beta_*^-(s_1; d_1, d_2)|$. Perhaps not so surprisingly, in the next case, namely when $s_1 - d_2 < 1 \leq s_1 + d_1 - d_2$, we obtain the exact same procedure, the only two difference being the condition in the first "if" statement (obviously) and in the body of the first nested "if" statement being $\beta_*^- \leftarrow \beta_*^- + s_1 + d_1 - d_2 - 1$.

The full procedure to compute $|\beta_*^-(s_1; d_1, d_2)|$ is presented in pseudocode 3.4.

Algorithm 3.4: Compute $|\beta_*^-(s_1; d_1, d_2)|$.

Input: n arrangement's size, d_1 and d_2 lengths of edges, $s_1 \leq n - d_1$ a fixed position, $d_1 \leq d_2$.
Output: The amount of pairs $(s_1, s_2) \in \beta(d_1, d_2)$ such that $s_2 < s_1$.

```
1  Function |β*⁻(s1; d1, d2)| is
2      β*⁻ ← 0
3      if 1 ≤ s1 − d2 then
4          β*⁻ ← β*⁻ + s1 + d1 − d2 − 2
5          if d1 < d2 then β*⁻ ← β*⁻ + d2 − d1 − 1
6      else if 1 ≤ s1 + d1 − d2 then
7          β*⁻ ← β*⁻ + s1 + d1 − d2 − 1
8          if d1 < d2 then β*⁻ ← β*⁻ + d2 − d1 − 1
9      else β*⁻ ← β*⁻ + s1 − 1
10     if n − d2 < s1 then
           // Substract overcounted cells
11         β*⁻ ← β*⁻ − s1 − (n − d2) − 1
12     return β*⁻
```

The total size of $\beta^-(d_1, d_2)$ is found by summing over all values of s_1 from 1 to $n - d_1$. The resulting function is encoded in algorithm 3.5. The sums corresponding to the functions at every "if" statement can be found in the same way they were found in equation 3.15.

Algorithm 3.5: Compute $|\beta^-(d_1, d_2)|$.

Input: n arrangement's size, d_1 and d_2 lengths of edges, $d_1 \leq d_2$.
Output: The amount of pairs $(s_1, s_2) \in \beta(d_1, d_2)$ such that $s_2 < s_1$.

Function $|\beta^-(d_1, d_2)|$ **is**
  $\beta^- \leftarrow 0$
  if $d_1 < d_2$ **then**
    if $1 + d_2 \leq n - d_1$ **then** $\beta^- \leftarrow \beta^- + (n - d_1)^2 - 5(n - d_1 - d_2) - d_2^2$
    if $d_2 \leq n - d_1$ **then** $\beta^- \leftarrow \beta^- + d_1(2d_2 - d_1 - 3)$
    else $\beta^- \leftarrow \beta^- + (n - d_2)(n - 2d_1 + d_2 - 3)$
  else
    if $1 + 2d_1 \leq n$ **then** $\beta^- \leftarrow \beta^- + n(n - 3) - d_1(2n - 6)$
    if $2d_1 \leq n$ **then** $\beta^- \leftarrow \beta^- + d_1(d_1 - 1)$
    else $\beta^- \leftarrow \beta^- + (n - d_2)(n - d_2 - 1)$
  return $\frac{1}{2}\beta^-$

Proposition 3.2. *For any $n \in \mathbb{N} > 0$, and any $1 \leq d_1 < d_2 \leq n$, computing $|\alpha(d_1, d_2)|$ and $|\beta(d_1, d_2)|$ can be done in constant time and constant space.*

Proof. The value $|\alpha(d_1, d_2)|$ can be computed as the sum of the results of equation 3.12 and algorithm 3.3. The value $|\beta(d_1, d_2)|$ can be computed as the sum of the results of equation 3.15 and algorithm 3.5. Both have constant time and space complexity. □

4 Variance of the number of crossings

We devote this section to studying the variance of the number of crossings in a graph over the space of uniformly random linear arrangements (rla). Formally, given the number of crossings for all the $n!$ possible linear arrangements, $\{C_G(\pi)\}_{\pi\in\Pi}$ the exact variance is defined as usual

$$\mathbb{V}_{rla}\left[\, C_G \,\right] = \frac{1}{n!}\sum_{\pi\in\Pi}(C_G(\pi) - \mathbb{E}_{rla}\left[\, X_C \,\right])^2. \tag{4.1}$$

In a previous work Alemany-Puig and Ferrer-i-Cancho [2] approached the problem of finding a simple arithmetic expression for the variance by analysing an equivalent formulation of the variance, i.e.

$$\mathbb{V}_{rla}\left[\, C_G \,\right] = \mathbb{E}_{rla}\left[\, (C_G - \mathbb{E}_{rla}\left[\, C_G \,\right])^2 \,\right]. \tag{4.2}$$

The authors realised that the squared difference yielded several "types of products". They first noticed that, since $C_G - \mathbb{E}_{rla}\left[\, C_G \,\right]$ is actually a sum of indicator variables (see equation 1.3) minus a constant, its square can be expressed as a double summation of products. Then

$$\mathbb{V}_{rla}\left[\, C_G \,\right] = \mathbb{E}_{rla}\left[\left(\sum_{\{st,uv\}\in Q}\left(c_\pi(st,uv) - \frac{1}{3}\right)\right)^2\right]$$

becomes

$$\mathbb{V}_{rla}\left[\, C_G \,\right] = \sum_{\{st,uv\}\in Q}\sum_{\{wx,yz\}\in Q}\mathbb{E}_{rla}\left[\left(c_\pi(st,uv) - \frac{1}{3}\right)\left(c_\pi(wx,yz) - \frac{1}{3}\right)\right] \tag{4.3}$$

by linearity of expectations, where π denotes a linear arrangement drawn uniformly at random from Π. The types of products are merely the classification of the edges st, uv, wx, yz according to two parameters. (1) τ the amount of edges shared between the pair $\{st, uv\}$ and $\{wx, yz\}$, and (2) ϕ the amount of vertices shared among all edges. A third parameter was required for only one of the types. Using the values of these parameters the authors found 9 different types, one of which had to be split into two types. These were summarised in [2, Table 2, page 17]. We say that two pairs $\{st, uv\} \in Q$ and $\{wx, yz\} \in Q$ are classified into a certain type if the values of τ, ϕ (and optionally the third parameter) correspond to that type. Using this, it can be immediately seen that the variance can be expressed as product of the amount of the amount of times each of these types appear in equation 4.3 and the value of the expectation of $c_\pi(st, uv)c_\pi(wx, yz)$. For the sake of compactness, they mapped each type into a single integer, from 0 to 8, and expressed the variance as

$$\mathbb{V}_{rla}\left[\, C_G \,\right] = \sum_{i=0}^{8} f_i(G)\,\mathbb{E}\left[\, \gamma_i \,\right] \tag{4.4}$$

where $f_i(G)$ represents the amount of pairs of elements of $Q(G) \times Q(G)$ classified into type i, and

$$\mathbb{E}\left[\, \gamma_i \,\right] = \mathbb{E}\left[\, c_\pi(st,uv)c_\pi(wx,yz) \,\right] - \frac{1}{9}$$

if, and only if, $\{st, uv\} \in Q$ and $\{wx, yz\} \in Q$ are classified into type i. These expectations do not depend on the graph G, only on the setting (here, the 1-dimensional case[1]). Notice that

$$f_i(G) = \sum_{q_1\in Q}\sum_{\substack{q_2\in Q:\\ type(q_1,q_2)=i}} 1.$$

[1] Other settings have been studied. See, for example, the work by J.W. Moon in [22], where the same problem is studied when the vertices of the graph are distributed uniformly at random over the surface of a sphere. Evidently, the value of the expectations change.

In [2], the authors showed that each type of product is actually a class of subgraph in G. For example, if a pair of two elements of Q is classified as type $\tau = 0$ and $\phi = 0$, meaning that they do not share edges or vertices, the graph is $\mathcal{L}_2 \oplus \mathcal{L}_2 \oplus \mathcal{L}_2 \oplus \mathcal{L}_2$. The authors came to the conclusion, then, that each $f_i(G)$ is directly proportional to the amount of subgraphs in G that are isomorphic to the graph representing the i-th type. They found the type of subgraph by both studying each type of product using the parameters described above, and algebraically. While doing so, they realised that each $f_i(G)$ not only counts a certain type of subgraph, but also a proportional amount, namely, they showed that $f_i(G) = a_i n_G(W_i)$, for all $i \in \{0, .., 8\}$, where $a_i \in \mathbb{N}$ and W_i is the graph of the i-th type of product. In equation 4.5 are summarised all $f_i(G)$ [2, Table 4], and in figure 4.1 are depicted all W_i.

$$
\begin{aligned}
f_0(G) = f_{00}(G) &= 6n_G(\mathcal{L}_2 \oplus \mathcal{L}_2 \oplus \mathcal{L}_2 \oplus \mathcal{L}_2), & f_1(G) = f_{24}(G) &= n_G(\mathcal{L}_2 \oplus \mathcal{L}_2),\\
f_2(G) = f_{13}(G) &= 2n_G(\mathcal{L}_3 \oplus \mathcal{L}_2), & f_3(G) = f_{12}(G) &= 6n_G(\mathcal{L}_2 \oplus \mathcal{L}_2 \oplus \mathcal{L}_2),\\
f_4(G) = f_{04}(G) &= 2n_G(\mathcal{C}_4), & f_5(G) = f_{03}(G) &= 2n_G(\mathcal{L}_5),\\
f_6(G) = f_{021}(G) &= 2n_G(\mathcal{L}_4 \oplus \mathcal{L}_2), & f_7(G) = f_{022}(G) &= 4n_G(\mathcal{L}_3 \oplus \mathcal{L}_3),\\
f_8(G) = f_{01}(G) &= 4n_G(\mathcal{L}_3 \oplus \mathcal{L}_2 \oplus \mathcal{L}_2). & & \qquad (4.5)
\end{aligned}
$$

The values of $\mathbb{E}_{rla}[\gamma_i]$ are constant rational values [2, Table 2]

$$
\begin{aligned}
&\mathbb{E}_{rla}[\gamma_0] = \mathbb{E}_{rla}[\gamma_{00}] = 0 \quad , \quad \mathbb{E}_{rla}[\gamma_1] = \mathbb{E}_{rla}[\gamma_{24}] = \frac{2}{9} \quad , \quad \mathbb{E}_{rla}[\gamma_2] = \mathbb{E}_{rla}[\gamma_{13}] = \frac{1}{18} \quad ,\\
&\mathbb{E}_{rla}[\gamma_3] = \mathbb{E}_{rla}[\gamma_{12}] = \frac{1}{45} \quad , \quad \mathbb{E}_{rla}[\gamma_4] = \mathbb{E}_{rla}[\gamma_{04}] = -\frac{1}{9}, \quad \mathbb{E}_{rla}[\gamma_5] = \mathbb{E}_{rla}[\gamma_{03}] = -\frac{1}{36},\\
&\mathbb{E}_{rla}[\gamma_6] = \mathbb{E}_{rla}[\gamma_{021}] = -\frac{1}{90}, \quad \mathbb{E}_{rla}[\gamma_7] = \mathbb{E}_{rla}[\gamma_{022}] = \frac{1}{180}, \quad \mathbb{E}_{rla}[\gamma_8] = \mathbb{E}_{rla}[\gamma_{01}] = 0.
\end{aligned}
\qquad (4.6)
$$

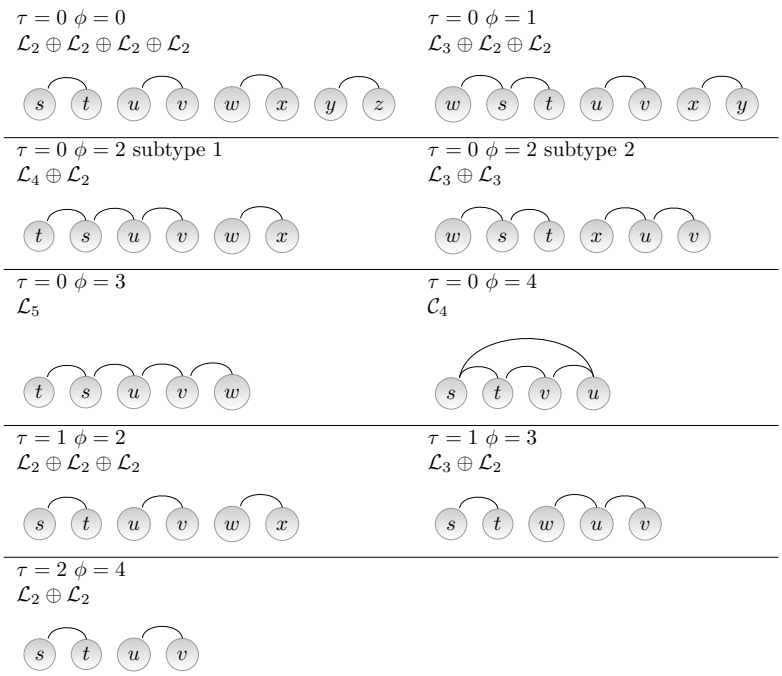


Figure 4.1: Types of subgraphs to be counted for each f_i in equations 4.5. Source [2, Figure 6].

A direct implementation of equation 4.4 is presented in pseudocode 4.1. Its cost is trivially $O\left(|Q|^2\right)$ since we have to enumerate all the elements in the Cartesian product $Q \times Q$.

Algorithm 4.1: Computing $\mathbb{V}_{rla}\left[C_G\right]$ in time $O\left(m^4\right)$.

Input: $G = (V, E)$ a graph.
Output: $\mathbb{V}_{rla}\left[C_G\right]$, the variance of the number of crossings C_G in general graphs.

```
1  Function VarianceC(G) is
2      F ← (0, ··· , 0) ∈ ℕ⁹ // F_i equals f_i(G).
3      for q_1 ∈ Q do
4          for q_2 ∈ Q do
               // Retrieve the type of pair (q_1, q_2).  w is the optional parameter.
5              (τ, φ, w) ← type of (q_1, q_2)
               // Map τ, φ, w into its corresponding integer in [0, 8].
6              i ← map(τ, φ, q)
7              F_i ← F_i + 1
8      for i = 0, ··· , 8 do
9          V ← V + E_rla[γ_i] · F_i
10     return V
```

The structure of this section is as follows. We first analyse the expectation of the variance in random graphs $G_{n,p} \in \mathcal{G}_{n,p}$. Then, in section 4.2 we devise algorithm for its exact computation in simple graphs. We first simplify equation 4.10 by giving equivalent formulations of some of its summations in the form of subgraph counting. These efforts yielded equation 4.17. Then, in section 4.3, we give mathematical expressions to efficiently evaluate the summations in equation 4.17 (see section 4.3.1) that allowed us to derive an efficient algorithm for evaluating it. In section 4.3.2 we put everything together and formalise the best algorithm that we could devise to compute the variance of the number of crossings in general graphs. This yielded a $O\left(n\langle k^2\rangle k_{max}\right)$-time algorithm. This algorithm has the particularity that computations can be reused. If that is the case, its space complexity increases, as it is explained in proposition 4.15, but it can be up to 6 times faster, according to the speed up measurements presented in table 4.1. Finally, we present, in section 4.3.3, an algorithm to compute $\mathbb{V}_{rla}\left[C_T\right]$, the variance of the number of crossings in the particular case of trees, and, in section 4.3.4, another to compute $\mathbb{V}_{rla}\left[C_F\right]$, the variance of the number of crossings in the particular case of trees. These two algorithms have both time complexity $O\left(n\right)$.

4.1 In random graphs

In section 3.1 we analysed the expected value of the number of crossings in a uniformly random linear arrangement of a graph $G_{n,p} \in \mathcal{G}_{n,p}$, formally $\mathbb{E}_{n,p}\left[\mathbb{E}_{rla}\left[C_{G_{n,p}}\right]\right]$. For the sake of comprehensiveness, we analyse here the expected value of the variance of the number of crossings in random graphs $G_{n,p}$, namely $\mathbb{E}_{n,p}\left[\mathbb{V}_{rla}\left[C_{G_{n,p}}\right]\right]$.

Analysing it directly using the equation for the variance in arbitrary simple graphs (see equation 4.10) proved to be too difficult. Thankfully, as explained at the beginning of this section, in [2] a different analysis of the variance was also given: in order to derive the aforementioned equation, they first gave $\mathbb{V}_{rla}\left[C_G\right]$ as the sum of the products of the amounts of 9 different types of subgraphs in G and the probability that their vertices cross when they are linearly arranged (minus a constant value). We presented this formalisation in equation 4.4.

Recall that the $f_i(G)$ represent the number of occurrences of the i-th type of subgraph in G, and $\mathbb{E}_{rla}\left[\gamma_i\right]$ is the probability that two edges classified into type i cross in a uniformly random linear arrangement (minus a constant value). These f_i and graphs are summarised in this work in equation 4.5 and figure 4.1.

Proposition 4.1. *Let $G_{n,p} \in \mathcal{G}_{n,p}$ be a random graph.*

$$\mathbb{E}_{n,p}\left[\,\mathbb{V}_{rla}\left[\,C_{G_{n,p}}\,\right]\,\right] = \frac{1}{15}\binom{n}{4}p^2(1-p)(p(n^2+n-10)+10). \tag{4.7}$$

Proof. Finding the expected value of the variance for a graph $G_{n,p} \in \mathcal{G}_{n,p}$ is reduced to finding the expected value of $f_i(G_{n,p})$. In symbols,

$$\begin{aligned}\mathbb{E}_{n,p}\left[\,\mathbb{V}_{rla}\left[\,C_{G_{n,p}}\,\right]\,\right] &= \mathbb{E}_{n,p}\left[\sum_{i=0}^{8} f_i(G)\,\mathbb{E}\left[\,\gamma_i\,\right]\right] \\ &= \sum_{i=0}^{8}\mathbb{E}_{n,p}\left[\,f_i(G)\,\right]\mathbb{E}\left[\,\gamma_i\,\right]\end{aligned}$$

by linearity of expectations and because the expectation of a constant value c is the constant value c. Therefore we have to calculate the expected amount of subgraphs isomorphic to W_i in $G_{n,p}$. For this, we make use of equation [5, Section VII]

$$\mathbb{E}_{n,p}\left[\,X_{W_i}\,\right] = n_{\mathcal{K}_n}(W_i)p^{|E(W_i)|}$$

where $X_{W_i} = n_{G_{n,p}}(W_i)$ is the number of subgraphs of $G_{n,p}$ isomorphic to W_i, and w_i is the graph corresponding to type i. In order to use this equation we need to know these values in complete graphs. Again, this is already done in [2, Table 6]. Using those values we obtain

$$\begin{aligned}
\mathbb{E}_{rla}\left[\,f_{00}(G_{n,p})\,\right] &= 630\binom{n}{8}p^4, & \mathbb{E}_{rla}\left[\,f_{24}(G_{n,p})\,\right] &= 3\binom{n}{4}p^2,\\
\mathbb{E}_{rla}\left[\,f_{13}(G_{n,p})\,\right] &= 60\binom{n}{5}p^3, & \mathbb{E}_{rla}\left[\,f_{12}(G_{n,p})\,\right] &= 90\binom{n}{6}p^3,\\
\mathbb{E}_{rla}\left[\,f_{04}(G_{n,p})\,\right] &= 6\binom{n}{4}p^4, & \mathbb{E}_{rla}\left[\,f_{03}(G_{n,p})\,\right] &= 120\binom{n}{5}p^4,\\
\mathbb{E}_{rla}\left[\,f_{021}(G_{n,p})\,\right] &= 360\binom{n}{6}p^4, & \mathbb{E}_{rla}\left[\,f_{021}(G_{n,p})\,\right] &= 360\binom{n}{6}p^4,\\
\mathbb{E}_{rla}\left[\,f_{021}(G_{n,p})\,\right] &= 1260\binom{n}{7}p^4.
\end{aligned}$$

Multiplying each of these values by their corresponding expectations in 4.6 we immediately achieve our goal

$$\begin{aligned}\mathbb{E}_{n,p}\left[\,\mathbb{V}_{rla}\left[\,C_{G_{n,p}}\,\right]\,\right] &= \sum_{i=0}^{8}\mathbb{E}_{rla}\left[\,\gamma_i\,\right]\cdot\mathbb{E}_{n,p}\left[\,f_i(G_{n,p})\,\right] \\ &= \frac{1}{15}\binom{n}{4}p^2(1-p)(p(n^2+n-10)+10).\end{aligned}$$

□

Figure 4.2 shows the value of function 4.7 for several values of n and p.

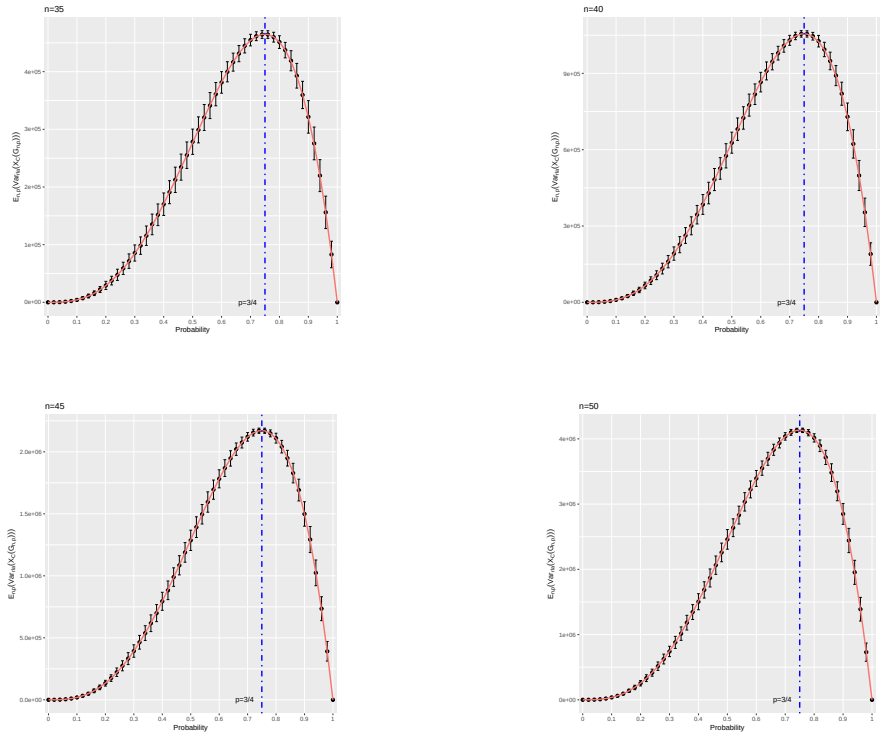


Figure 4.2: Value of $\mathbb{E}_{n,p}\left[\mathbb{V}_{rla}\left[G_{n,p}\right]\right]$ (red line, see equation 4.7) for different values of n and p. For each value of p we generated 500 random graphs and plotted the value $\mathbb{V}_{rla}\left[C_{G_{n,p}}\right]$ for each of these graphs. The dashed vertical blue line denotes the value of p for which the expected variance is maximised (see result in 4.9).

4.1.1 Maximum expected variance

Proposition 4.2. *Let $G_{n,p} \in \mathcal{G}_{n,p}$ be a random graph. The maximum expected variance $\mathbb{V}_{rla}\left[C_{G_{n,p}}\right]$ is found at $p = 3/4$ and has value*

$$\mathbb{E}_{n,3/4}\left[\mathbb{V}_{rla}\left[C_{G_{n,3/4}}\right]\right] = \frac{3}{1280}\binom{n}{4}(3n^2+3n+10). \tag{4.8}$$

Proof. Since now we know how to estimate the variance of a graph $G_{n,p} \in \mathcal{G}_{n,p}$ (see equation 4.7), we can find out the value of p that maximises it. We do so by taking the partial derivative with respect to p

$$\frac{\partial}{\partial p}\mathbb{E}_{n,p}\left[\mathbb{V}_{rla}\left[G_{n,p}\right]\right] = \frac{1}{15}\binom{n}{4}\left((n^2+n-10)(3p^2-4p^3)+20p-30p^2\right) = 0.$$

We can rearrange it to obtain

$$-4Np^3+(3N-30)p^2+20p = 0$$

where $N = n^2+n-10$. One trivial solution is $p = 0$. The others can be found by solving the quadratic equation resulting from dividing the left hand side by p:

$$p = \frac{3N-30N \pm \sqrt{9N^2+140N+900}}{8N}.$$

After some more algebraic manipulations we obtain

$$p = \frac{3n^2+3n-60 \pm \sqrt{n(n+1)(3n-5)(3n+8)+400}}{8n^2+8n-10}.$$

Let $S^+(n)$ and $S^-(n)$ be the value of p as a function of n when the square root has positive and negative sign, respectively. Taking limits, we find that only $S^+(n)$ leads to a non-zero solution:

$$\lim_{n\to+\infty} S^+(n) = \lim_{n\to+\infty} \frac{3n^2+3n+\sqrt{9n^2}}{8n^2+8n} = \frac{3}{4}. \tag{4.9}$$

Therefore, we can expect to find the random graph that maximises the variance around the value $p = 3/4$. The maximum expected variance in a graph $G_{n,p} \in \mathcal{G}_{n,p}$ is, then

$$\mathbb{E}_{n,3/4}\left[\mathbb{V}_{rla}\left[C_{G_{n,3/4}}\right]\right] = \frac{3}{1280}\binom{n}{4}(3n^2+3n+10).$$

□

Figure 4.2 remarks the value $p = 3/4$ with a thick blue line.

4.2 Simplifying the expression of the variance

Although equation 4.4 can be evaluated in time $O\left(m^4\right)$ with a direct algorithm, said equation is a big improvement over equation 4.1, for which a direct algorithm would have $O\left(n!\right)$-time complexity, and takes us one step closer to an exact computation of the variance. However, further algebraic analyses by the same authors showed that the variance can be expressed as [2, Equation 95]

$$\begin{aligned}
\mathbb{V}_{rla}\left[C_G\right] = {} & \frac{2}{45}(m+2)|Q| - \frac{1}{180}n_G(\mathcal{L}_5) - \frac{2m+7}{180}n_G(\mathcal{L}_4) - \frac{3}{45}n_G(\mathcal{C}_4) + \frac{1}{90}K_G \\
& - \frac{1}{60}\sum_{\{st,uv\}\in Q}(k_s(a_{tu}+a_{tv}) + k_t(a_{su}+a_{sv}) + k_u(a_{sv}+a_{vt}) + k_v(a_{su}+a_{tu})) \\
& + \frac{1}{180}\sum_{\{st,uv\}\in Q}(a_{su}+a_{sv}+a_{tu}+a_{tv})(k_s+k_t+k_u+k_v) \\
& + \frac{1}{180}\sum_{\{st,uv\}\in Q}(k_s+k_t)(k_u+k_v) - \frac{1}{90}\sum_{\{st,uv\}\in Q}(k_sk_t+k_uk_v) \\
& + \frac{1}{30}\sum_{\{st,uv\}\in Q}(a_{tu}+a_{sv})(a_{tv}+a_{su}) \\
& + \frac{1}{90}\sum_{\{st,uv\}\in Q}\left(\sum_{w_s\in\Gamma(s,-stuv)} a_{tw_s} + \sum_{w_u\in\Gamma(u,-stuv)} a_{vw_u}\right).
\end{aligned} \tag{4.10}$$

where $K_G = \sum_{\{st,uv\}\in Q(G)}(k_s+k_t+k_u+k_v)$. Interestingly, as shown by Alemany-Puig and Ferrer-i-Cancho [2, Propositions 4.1, 4.2, 4.3], we have that

$$\begin{aligned}
n_G(\mathcal{L}_4) &= \sum_{st,uv\in Q}(a_{su}+a_{sv}+a_{tu}+a_{tv}), \\
n_G(\mathcal{L}_5) &= \sum_{st,uv\in Q}\left(\sum_{w\in\Gamma(s,-stuv)}(a_{uw}+a_{vw}) + \sum_{w\in\Gamma(t,-stuv)}(a_{uw}+a_{vw})\right), \\
n_G(\mathcal{C}_4) &= \frac{1}{2}\sum_{st,uv\in Q}(a_{su}a_{tv}+a_{sv}a_{tu}).
\end{aligned} \tag{4.11}$$

These three equations allow us to immediately write an algorithm that evaluates equation 4.10 by enumerating the elements of Q while computing the terms in each summation. However, this is not

efficient since $|Q| = O\left(m^2\right)$ and we would need the adjacency matrix of the graph, which takes up a space of n^2. Even though, this is yet another improvement since $\mathbb{V}_{rla}\left[C_G\right]$ can now be computed in time $O\left(m^2 k_{max}\right)$ with a direct algorithm. The details are omitted due to not being the goal of this section. They also showed that [2, Proposition 4.4]

$$K_G = \sum_{\{st,uv\}\in Q} (k_s + k_t + k_u + k_v) = (m+1)n\langle k^2\rangle - n\langle k^3\rangle - 2L_G, \tag{4.12}$$

where

$$L_G = \sum_{st\in E} k_s k_t.$$

We reference this equation because it is needed to design efficient algorithms for the computation of $\mathbb{V}_{rla}\left[C_G\right]$.

Equation 4.10 gives a really interesting hindsight on the variance. First, notice that the graphs that represent each type of product (all depicted in figure 4.1) no longer play a role in the evaluation of the variance (except for $\mathcal{C}_4$, type 04) and it unveils the role played by $\mathcal{L}_4$ and $\mathcal{L}_5$. Moreover, as showed in [2], the variance can be directly simplified to [2, Equation 96]

$$\begin{aligned}
\mathbb{V}_{rla}\left[C_T\right] = {} & \frac{2}{45}(m+2)|Q| - \frac{1}{180}n_T(\mathcal{L}_5) - \frac{2m+7}{180}n_T(\mathcal{L}_4) + \frac{1}{90}K_T \\
& - \frac{1}{60}\sum_{\{st,uv\}\in Q} (k_s(a_{tu}+a_{tv}) + k_t(a_{su}+a_{sv}) + k_u(a_{sv}+a_{tv}) + k_v(a_{su}+a_{tu})) \\
& + \frac{1}{180}\sum_{\{st,uv\}\in Q} (a_{su}+a_{sv}+a_{tu}+a_{tv})(k_s+k_t+k_u+k_v) \\
& + \frac{1}{180}\sum_{\{st,uv\}\in Q} (k_s+k_t)(k_u+k_v) - \frac{1}{90}\sum_{\{st,uv\}\in Q} (k_s k_t + k_u k_v),
\end{aligned} \tag{4.13}$$

for trees. Needless to say that in equation 4.10 we use $Q = Q(G)$, whereas in equation 4.13, $Q = Q(T)$.

Although equations 4.10 and 4.13 are a really good improvement over equation 4.4, when expressed in this way we can see that some of the summations do not allow for efficient implementations. For example, there does not seem to be much hope in obtaining an efficient, direct evaluation of the summation

$$\sum_{\{st,uv\}\in Q} (a_{tu}+a_{sv})(a_{tv}+a_{su})$$

since it seems that it requires the enumeration of all the elements of Q.

In this section we show how two of the summations in equation 4.10 are directly related to the problem of counting a particular subgraph in a larger graph G. These two results allow us to obtain more efficient algorithms, described in the coming sections. Each summation counts the amount of a different type of graph, both depicted in figure 4.3, one of them (4.3(a)) being the paw graph [29]. These two results are formalised in propositions 4.3 and 4.4.

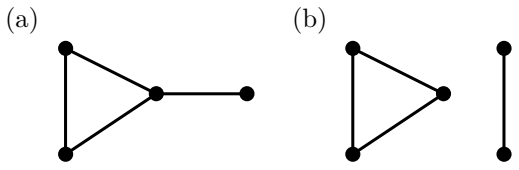


Figure 4.3: Two subgraphs whose amount is counted in equation 4.10 via two of its summations, as proved in (a) proposition 4.3 for the graph paw, and in (b) proposition 4.4 for $\mathcal{C}_3 \oplus \mathcal{L}_2 = \overline{\mathcal{K}_{2,3}}$.

The first proposition proves that one of the summations counts the amount of subgraphs isomorphic to the graph depicted in figure 4.3(a). The second proves that one of the summations counts the amount of subgraphs isomorphic to the graph depicted in figure 4.3(b).

Proposition 4.3. *Let $G = (V, E)$ be a graph and $Q = Q(G)$.*

$$n_G(Z) = \sum_{\{st,uv\}\in Q} (a_{tu} + a_{sv})(a_{tv} + a_{su}) \tag{4.14}$$

where Z denotes the graph paw, depicted in figure 4.3(a).

Proof. The proof is simple. We first show that the term in the summation

$$(a_{tu} + a_{sv})(a_{tv} + a_{su}) = a_{tu}a_{tv} + a_{tu}a_{su} + a_{sv}a_{tv} + a_{sv}a_{su}$$

counts the amount of subgraphs isomorphic to Z in G that we can form given a fixed $\{st, uv\} \in Q$. This can be easily seen in figure 4.4 where, for a given $q = \{st, uv\} \in Q$, are shown all possible labelled graphs, $\mathcal{Z}(q)$, isomorphic to Z, that can be made with st and uv assuming the existence of the pairs of edges in the summation $a_{tu}a_{tv} + a_{tu}a_{su} + a_{sv}a_{tv} + a_{sv}a_{su}$. This means that when counting how many of these pairs of edges exist we are actually counting how many subgraphs isomorphic to Z exist that have these four vertices.

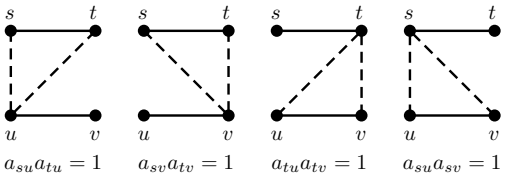
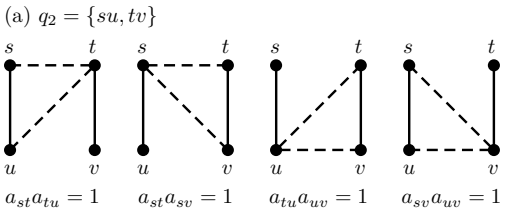


Figure 4.4: All possible graphs paw (figure 4.3(a)), that can be made with $\{st, uv\} \in Q$ given the adjacencies at the bottom of each graph.

Now we prove the claim in this proposition by contradiction. Since we know that the term inside the summation counts all labelled Z that can be made with every element of Q, the claim can only be false for two reasons: in the whole summation of equation 4.14 some Z is not counted, and/or some of these Z is counted more than once.

Firstly, it is clear that all Z are counted at least once. If one was not counted then the element of Q we can make with its vertices would not be in Q. This cannot happen by definition of Q.

Secondly, none is counted more than once. Let $\mathcal{Z}(q_1)$ be the set of labelled graphs a fixed element $q_1 \in Q$ is mapped to. If any $Z \in \mathcal{Z}(q_1)$ is counted twice then there exists a different $q_2 \in Q$ such that $\mathcal{Z}(q_1) \cap \mathcal{Z}(q_2) \neq \emptyset$. It is obvious that we need q_2 to have the same vertices as q_1. Therefore, if $q_1 = \{st, uv\}$ then we must have either $q_2 = \{su, tv\}$ or $q_2 = \{sv, tu\}$. All the graphs that can be made with both configurations are depicted in figure 4.5. None of these graphs are in $\mathcal{Z}(q_1)$.



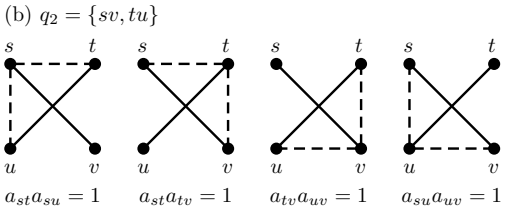


Figure 4.5: All possible graphs paw (figure 4.3(a)) that can be made with (a) $\{su, tv\} \in Q$, and (b) $\{sv, tu\} \in Q$ given the adjacencies indicated at the bottom of each graph.

□

Proposition 4.4. *Let $G = (V, E)$ be a graph and $Q = Q(G)$.*

$$n_G(Y) = \frac{1}{3} \sum_{\{st,uv\} \in Q} \left(\sum_{w_s \in \Gamma(s, -stuv)} a_{tw_s} + \sum_{w_u \in \Gamma(u, -stuv)} a_{vw_u} \right) \tag{4.15}$$

where $Y = \mathcal{C}_3 \oplus \mathcal{L}_2$ is depicted in figure 4.3(b).

Proof. We use $Y = \mathcal{C}_3 \oplus \mathcal{L}_2$ for the sake of brevity. Similarly as in proposition 4.3 we first show that the inner summation of the right hand side of equation 4.14, i.e.,

$$\sum_{w_s \in \Gamma(s, -stuv)} a_{tw_s} + \sum_{w_u \in \Gamma(u, -stuv)} a_{vw_u} \tag{4.16}$$

counts the amount of graphs isomorphic to Y that can be made using the edges of a fixed element $q = \{st, uv\} \in Q$, that we denote as $\mathcal{Y}(q)$. None of these graphs are repeated and are illustrated in figure 4.6. Given a fixed $\{st, uv\} \in Q$ the vertices w_s denote, in the figure, the common neighbours of s and t different from s, t, u, v, namely $w_s \in \{x \in \Gamma(s, -stuv) \mid a_{tx} = 1\}$. Likewise for w_u. For every w_s and w_u we have a different Y.

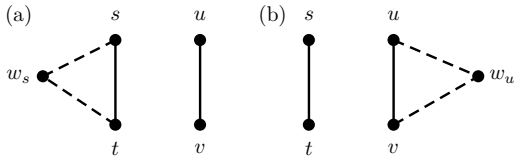


Figure 4.6: All the labelled graphs isomorphic to $\mathcal{C}_3 \oplus \mathcal{L}_2$ (shown in figure 4.3(b)) that can be formed using a fixed element $\{st, uv\} \in Q$. In this figure, w_s (w_u) is one of the common neighbours of s and t (u and v) different from s, t, u, v. The graphs in (a) correspond to the first summation of equation 4.16 and the graphs in (b) correspond to the second.

Evidently all subgraphs isomorphic to Y are counted: if one is not counted then the three elements of Q we can make with its vertices would not be in Q. This cannot happen by definition. Notice that if Y has vertices $V(Y) = \{c_1, c_2, c_3, e_1, e_2\}$, where the c_i are the vertices of $\mathcal{C}_3$ and e_i the vertices of $\mathcal{L}_2$ then we have

$$\{c_1c_2, e_1e_2\}, \{c_1c_3, e_1e_2\}, \{c_2c_3, e_1e_2\} \in Q.$$

It remains to show that every element is counted exactly three times. This is hinted by the previous observation that from any given Y we can make three elements $q \in Q$. Let $q_1 = \{st, uv\} \in Q$ be fixed, and let $\mathcal{Y}(q_1)$ be the set of subgraphs in G isomorphic to Y with the edges st and uv. The graphs in $\mathcal{Y}(q_1)$ can only be counted again, in the outer summation of equation 4.15, for those elements $q_2 \in Q$

such that $q_2 = \{sw_s, uv\}$, $q_2 = \{tw_s, uv\}$, $q_2 = \{st, uw_u\}$, and $q_2 = \{st, vw_u\}$, where w_s and w_u are defined as before. Notice that for any of these elements we have $|\mathcal{Y}(q_1) \cap \mathcal{Y}(q_2)| = 1$. Now, for all graphs in $\mathcal{Y}(q_1)$ that have st as an edge of its $\mathcal{C}_3$ the only $q_2 \in Q$ that count the same graphs (in the summation) are the first two: $q_2 = \{sw_s, uv\}$ and $q_2 = \{tw_s, uv\}$. For those graphs in $\mathcal{Y}(q_1)$ where uv is an edge of $\mathcal{C}_3$ the only $q_2 \in Q$ that count the same graphs are the last two: $q_2 = \{st, uw_u\}$ and $q_2 = \{st, vw_u\}$. Hence the factor 3 in the right hand side of equation 4.15. □

Using the results in propositions 4.3 and 4.4, the variance of the number of crossings in the space of random linear arrangements, $\mathbb{V}_{rla}[C_G]$, can be rewritten as

$$\begin{aligned}
\mathbb{V}_{rla}[C_G] = {} & \frac{2}{45}(m+2)|Q| - \frac{1}{180}n_G(\mathcal{L}_5) - \frac{2m+7}{180}n_G(\mathcal{L}_4) - \frac{3}{45}n_G(\mathcal{C}_4) + \frac{1}{90}K_G \\
& + \frac{1}{30}n_G(\mathcal{C}_3 \oplus \mathcal{L}_2) + \frac{1}{30}n_G(Z) \\
& - \frac{1}{60}\sum_{\{st,uv\}\in Q}(k_s(a_{tu}+a_{tv}) + k_t(a_{su}+a_{sv}) + k_u(a_{sv}+a_{vt}) + k_v(a_{su}+a_{tu})) \\
& + \frac{1}{180}\sum_{\{st,uv\}\in Q}(a_{su}+a_{sv}+a_{tu}+a_{tv})(k_s+k_t+k_u+k_v) \\
& + \frac{1}{180}\sum_{\{st,uv\}\in Q}(k_s+k_t)(k_u+k_v) - \frac{1}{90}\sum_{\{st,uv\}\in Q}(k_sk_t + k_uk_v)
\end{aligned} \tag{4.17}$$

where Z is the graph paw (see figure 4.3(a)) and K_G denotes the expression in equation 4.12. This equivalent way of expressing the variance highlights more clearly which are the terms that vanish in trees (those involving cycles). Thus, equation 4.13 can be understood more easily.

4.3 Algorithms to compute the variance

Some of the summations in equation 4.17 can be simplified into simpler expressions that can be evaluated without the set Q. This is important since the enumeration of such a set is costly, specially when its size is close to $O(m^2)$.

We first give the two simplest results where we simplify the last two summations in equation 4.17. Then, in section 4.3.1 we derive expressions that can be used to derive suitable algorithms, for our purposes, for counting the subgraphs $n_G(...)$ that appear in equation 4.17. This could be done using the work by Hocevar *et. al.*, where they developed algorithms to compute all graphs of 4 and 5 vertices by solving a system of linear equations. Our case, however, is much simpler. The remaining two summations are closely related to the problem of counting subgraphs and are simplified at the end of 4.3.1. We then continue to detail the algorithms for computing the variance in general graphs, i.e. $\mathbb{V}_{rla}[C_G]$ by evaluating equation 4.17, in section 4.3.2. Finally, we give a $O(n)$-time algorithm for computing the variance in trees in section 4.3.3.

Now follow mathematical expressions used to solve the problem of counting subgraphs. In most of these expressions we use the notation $\xi(s)$ and $c(s,t)$. The former is the sum of the degrees of the neighbours of each vertex, and the latter is the set of common neighbours between two vertices s and t. Formally,

$$\begin{aligned}
\xi(s) &= \sum_{u\in\Gamma(s)} k_u, \quad \forall s \in V \\
c(s,t) &= \Gamma(s) \cap \Gamma(t).
\end{aligned}$$

The simplest simplifications are gathered in the following two propositions.

Proposition 4.5. *Let $G = (V, E)$ be a graph and $Q = Q(G)$,*

$$\sum_{\{st,uv\}\in Q} (k_s k_t + k_u k_v) = \sum_{st\in E} k_s k_t (m - k_s - k_t + 1). \tag{4.18}$$

Proof. The proof is simple:

$$\begin{aligned}\sum_{\{st,uv\}\in Q} (k_s k_t + k_u k_v) &= \sum_{\{st,uv\}\in Q} k_s k_t + \sum_{\{st,uv\}\in Q} k_u k_v \\ &= \frac{1}{2}\left[\sum_{st\in E}\sum_{uv\in E(G_{-st})} k_s k_t + \sum_{uv\in E}\sum_{st\in E(G_{-uv})} k_u k_v\right] \\ &= \sum_{st\in E}\sum_{uv\in E(G_{-st})} k_s k_t = \sum_{st\in E} k_s k_t \sum_{uv\in E(G_{-st})} 1.\end{aligned}$$

The size of $E(G_{-st})$ equals the amount of edges of the graph induced from the removal of vertices s and t. This is $|E(G_{-st})| = m - k_s - k_t + 1$. □

Now follows the second simplification.

Proposition 4.6. *Let $G = (V, E)$ be a graph and $Q = Q(G)$.*

$$\sum_{\{st,uv\}\in Q} (k_s + k_t)(k_u + k_v) = \frac{1}{2}\sum_{st\in E}\left[(k_s + k_t)\left(n\langle k^2\rangle - \xi(s) - \xi(t) - k_s(k_s - 1) - k_t(k_t - 1)\right)\right]. \tag{4.19}$$

Proof. The proof is also straightforward. We start by noticing that

$$\sum_{\{st,uv\}\in Q} (k_s + k_t)(k_u + k_v) = \frac{1}{2}\sum_{st\in E}(k_s + k_t)\sum_{uv\in E(G_{-st})}(k_u + k_v).$$

Simplifying the inner sum is a tedious task. An equivalent, more self-explanatory sum iterates over the set of edges in E that are not incident to any of the edges s or t, or the edge $\{s, t\}$ itself. Formally, for a fixed edge $\{s, t\}$, the summation iterates over the set

$$E \setminus \big(\quad \{\{u, s\} \in E \ : \ u \in V \setminus \{t\}\} \cup \{\{u, t\} \in E \ : \ u \in V \setminus \{s\}\} \cup \{\{s, t\}\} \quad \big).$$

Then, for a fixed edge $\{s, t\} \in E$, this leads to

$$\begin{aligned}\sum_{uv\in E(G_{-st})} (k_u + k_v) &= \sum_{\substack{us\in E\\ u\neq t}} k_u(k_u - 1) + \sum_{\substack{ut\in E\\ u\neq s}} k_u(k_u - 1) + \sum_{\substack{u\in V\setminus\{s,t\}\\ us\notin E\\ ut\notin E}} k_u^2 \\ &= -k_t(k_t - 1) - k_s(k_s - 1) + \sum_{us\in E} k_u(k_u - 1) + \sum_{ut\in E} k_u(k_u - 1) + \sum_{\substack{u\in V\setminus\{s,t\}\\ us\notin E\\ ut\notin E}} k_u^2.\end{aligned}$$

The last summation can be further simplified

$$\begin{aligned}\sum_{\substack{u\in V\setminus\{s,t\}\\ us\notin E\\ ut\notin E}} k_u^2 &= -k_s^2 - k_t^2 + \sum_{u\in V} k_u^2 - \sum_{\substack{us\in E\\ u\neq t}} k_u^2 - \sum_{\substack{ut\in E\\ u\neq s}} k_u^2 \\ &= \sum_{u\in V} k_u^2 - \sum_{us\in E} k_u^2 - \sum_{ut\in E} k_u^2.\end{aligned}$$

After successive, and rather simple, algebraic manipulations we reach the expression in equation 4.19. □

The previous two results are quite simple and are not simplified any further (not even in the case of trees).

4.3.1 Counting subgraphs

The simpler expression for $\mathbb{V}_{rla}[C_G]$ (see equation 4.17) contains several countings of subgraphs, either explicitly (i.e., $n_G(\mathcal{L}_4)$, $n_G(\mathcal{L}_5)$, $n_G(\mathcal{C}_4)$, $n_G(Z)$ and $n_G(Y)$) or implicitly. Recall that Z is the paw graph (see figure 4.3(a)), and $Y = \mathcal{C}_3 \oplus \mathcal{L}_2 = \overline{\mathcal{K}_{2,3}}$ (see figure 4.3(b)). Only two of the summations of the simpler expression for the variance fall in the second category, i.e.

$$\sum_{\{st,uv\}\in Q} (k_s(a_{tu}+a_{tv}) + k_t(a_{su}+a_{sv}) + k_u(a_{sv}+a_{vt}) + k_v(a_{su}+a_{tu}))\,,$$

and

$$\sum_{\{st,uv\}\in Q} (a_{su}+a_{sv}+a_{tu}+a_{tv})(k_s+k_t+k_u+k_v).$$

In this section we give the necessary mathematical expressions to compute the explicit and implicit subgraph counting. Regarding the former, we are aware that there are previous works on their computation. For example, Movarraei [23] gave expressions for computing $n_G(\mathcal{L}_4)$ and $n_G(\mathcal{L}_5)$

$$n_G(\mathcal{L}_4) = \frac{1}{2}\sum_{i\neq j}(a_{ij}^{(3)} - (2k_j-1)a_{ij}), \tag{4.20}$$

$$n_G(\mathcal{L}_5) = \frac{1}{2}\left[\sum_{i\neq j}(a_{ij}^{(4)} - 2a_{ij}^{(2)}(k_j-a_{ij})) - \sum_{i=1}^{n}\left((2k_i-1)a_{ii}^{(3)} + 6\binom{k_i}{3}\right)\right], \tag{4.21}$$

where $A^x = [a_{ij}^{(x)}]$ is the x-th power of the adjacency matrix of the graph, G. Alon *et. al.*[3] gave expressions to count cycles of a given length. For example, they showed that

$$n_G(\mathcal{C}_3) = \frac{1}{6}tr(A^3), \quad n_G(\mathcal{C}_4) = \frac{1}{8}\left[\mathrm{tr}\left(A^4\right) - 4n_G(H_2) - 2n_G(H_1)\right]$$

where

$$n_G(H_2) = \sum_{u\in V}\binom{k_u}{2}, \quad n_G(H_1) = m.$$

However, none of these expressions are suitable for our purposes since they require the adjacency matrix of the graph and the computation of its powers, which would yield an algorithm with time complexity $O(n^\omega)$ and space complexity $O(n^2)$, where ω is the exponent of matrix multiplication. When counting cycles, it is possible not to use the whole matrix since we only need the trace of A^x which can be computed as the sum of the eigenvalues each raised to the x power, but we still need it for $n_G(\mathcal{L}_4)$ and $n_G(\mathcal{L}_5)$, even in trees. We, on the other hand, tackle the same problem but from a point of view where there is no need to use the adjacency matrix, and prove to be extremely useful for our purposes, and even better than those by Alon *et. al.*, and Movarraei. They are by no means the simplest, but they can all be easily evaluated at the same time, i.e. the expressions presented here can all be evaluated using the same traversal of the graph (a simple iteration over the set of edges).

Explicit subgraph counting The results given in this subsection tackle the problem of subgraph counting from a combinatorial point of view. They are sorted in increasing order of complexity. Therefore, a good starting point is an expression to count the cycles of 4 vertices, namely $\mathcal{C}_4$. A very simple combinatorial approach to counting cycles of 4 vertices is the following.

Proposition 4.7. *Let $G = (V, E)$ be a graph.*

$$n_G(\mathcal{C}_4) = \frac{1}{4}\sum_{st\in E}\sum_{u\in\Gamma(t)\setminus\{s\}}|c(s,u)| = \frac{1}{8}\left[tr\left(A^4\right) - 4\sum_{u\in V}\binom{k_u}{2} - 2m\right] = \frac{1}{8}\left[tr\left(A^4\right) - 2m - 4n_G(\mathcal{L}_3)\right]. \tag{4.22}$$

Proof. It is easy to see that the leading factor $1/4$ is needed since all edges of every $\mathcal{C}_4$ are visited four times each. The rest is trivial. The second equality was proven by Alon *et. al.*[3], and the third by Harary *et. al.*[17]. □

The expression for $n_G(Z)$, where Z is the paw graph, depicted in figure 4.3(a), is also quite simple.

Proposition 4.8. *Let $G = (V, E)$ be a graph, and let Z be the paw graph, depicted in figure 4.3(a). Then,*

$$n_G(Z) = \sum_{st\in E}\sum_{u\in c(s,t)}(k_u - 2) = \frac{1}{2}\left[\sum_{i=1}^{n} a_{ii}^{(3)}(k_i - 2)\right] \tag{4.23}$$

Proof. The proof of the first equality is easy. Without loss of generality, consider $u \in c(s,t)$ where $\{s,t\} \in E$. The vertices s, t, u induce a cycle of 3 vertices in G. The graphs isomorphic to Z with cycle s, t, u are those where the fourth vertex v is connected to u and $v \neq s, t$. Notice that there are $k_u - 2$ of such vertices. The second equality was proven by Alon *et. al.*[3]. □

Now follows a similar result to the previous. We aim at counting the amount of pairs of disjoint $\mathcal{C}_3$ and $\mathcal{L}_2$, namely the amount of subgraphs isomorphic to Y.

Proposition 4.9. *Let $G = (V, E)$ be a graph, and let $Y = \mathcal{C}_3 \oplus \mathcal{L}_2 = \overline{\mathcal{K}_{2,3}}$, depicted in figure 4.3(b).*

$$n_G(Y) = \frac{1}{3}\sum_{st\in E}\sum_{u\in c(s,t)}(m - k_s - k_t - k_u + 3) \tag{4.24}$$

Proof. Likewise, for any $\{s,t\} \in E$ and $u \in c(s,t)$, the vertices s, t, u induce a cycle in G of 3 vertices. It remains to count how many edges are independent of that cycle, i.e., edges whose endpoints do not coincide with s, t or u. Then, from the whole set of m edges we have to substract those connected to s, t or u. However, a correction term is needed since each edge of the cycle would be substracted twice. Hence the extra term $+3$. Finally, we need to divide the result of the summation by 3 so as to take symmetries into account. □

Now we present the method that we use to count the subgraphs isomorphic to linear trees of 4 vertices in the algorithm for the computation of the variance in general graphs.

Proposition 4.10. *Let $G = (V, E)$ be a graph.*

$$n_G(\mathcal{L}_4) = m - n\langle k^2\rangle + \frac{1}{2}\sum_{st\in E}(\xi(s) + \xi(t)) - \sum_{st\in E}|c(s,t)|. \tag{4.25}$$

Proof. Consider three vertices s,t,u inducing a path of 3 vertices in G: (s, t, u). We can count all induced subgraphs $\mathcal{L}_4$ that start with vertices s, t, u and finish at $v \in \Gamma(u)$ by counting how many v are different from s and t in the neighbourhood of u, $\Gamma(u)$. Then,

$$n_G(\mathcal{L}_4) = \frac{1}{2}\sum_{s\in V}\sum_{t\in\Gamma(s)}\sum_{u\in\Gamma(t)\setminus\{s\}}\sum_{v\in\Gamma(u)\setminus\{s,t\}} 1.$$

We can easily replace the inner-most summation with the expression $k_u - 1 - a_{su}$. This expression, when summed over the vertices $u \in \Gamma(t) \setminus \{s\}$, can be simplified further,

$$n_G(\mathcal{L}_4) = \frac{1}{2} \sum_{s \in V} \sum_{t \in \Gamma(s)} (\xi(t) - (k_s + k_t) + 1 - |c(s,t)|).$$

This is simple enough to see that the following expression is equivalent to the previous

$$n_G(\mathcal{L}_4) = \frac{1}{2} \sum_{st \in E} (\xi(s) + \xi(t) - 2(k_s + k_t) + 2 - 2|c(s,t)|).$$

Obtaining the expression in equation 4.25 is now straightforward. □

The following proposition gives yet another combinatorial approach, this time for counting linear trees of 5 vertices.

Proposition 4.11. *Let $G = (V, E)$ be a graph.*

$$n_G(\mathcal{L}_5) = \frac{1}{2} \sum_{s \in V} \sum_{t \in \Gamma(s)} \sum_{u \in \Gamma(s) \setminus \{t\}} ((k_t - 1 - a_{ut})(k_u - 1 - a_{ut}) + 1 - |c(t,u)|). \tag{4.26}$$

Proof. The proof is also straightforward and similar to the proof in proposition 4.10.

$$\begin{aligned} n_G(\mathcal{L}_5) &= \frac{1}{2} \sum_{s \in V} \sum_{t \in \Gamma(s)} \sum_{u \in \Gamma(s) \setminus \{t\}} \sum_{v \in \Gamma(t) \setminus \{s,u\}} \sum_{w \in \Gamma(u) \setminus \{s,t,v\}} 1 \\ &= \frac{1}{2} \sum_{s \in V} \sum_{t \in \Gamma(s)} \sum_{u \in \Gamma(s) \setminus \{t\}} \sum_{v \in \Gamma(t) \setminus \{s,u\}} (k_u - 1 - a_{ut} - a_{uv}) \\ &= \frac{1}{2} \sum_{s \in V} \sum_{t \in \Gamma(s)} \sum_{u \in \Gamma(s) \setminus \{t\}} ((k_t - 1 - a_{ut})(k_u - 1 - a_{ut}) + 1 - |c(t,u)|). \end{aligned}$$

□

Implicit subgraph counting Now we deal with the remaining two summations of equation 4.17, which do not directly count subgraphs, but a measure which depends on a type of subgraph itself. This subgraph is in both cases $\mathcal{L}_4$. This simple fact can be seen with the help of equation 4.11. which shows that the sum over all $\{st, uv\} \in Q(G)$ of $a_{su} + a_{sv} + a_{tu} + a_{tv}$ equals the amount of $\mathcal{L}_4$ in G. This was proved in [2, Proposition 4.1]. Propositions 4.12 and 4.13 rely on the fact that for a fixed $\{st, uv\} \in Q(G)$ we have that $a_{su} + a_{sv} + a_{tu} + a_{tv} \leq 4$, an immediate conclusion of [2, Proposition 4.1] (see equation 4.11).

Proposition 4.12. *Let $G = (V, E)$ be a graph and $Q = Q(G)$. The expression*

$$\sum_{\{st,uv\} \in Q} (k_s(a_{tu} + a_{tv}) + k_t(a_{su} + a_{sv}) + k_u(a_{sv} + a_{vt}) + k_v(a_{su} + a_{tu})) \tag{4.27}$$

is equivalent to

$$\frac{1}{2} \sum_{s \in V} \sum_{t \in \Gamma(s)} ((k_t - 1)(\xi(s) - k_t) + k_s(\xi(t) - k_s - k_t + 1) - 2k_s|c(s,t)|)\,. \tag{4.28}$$

Proof. If we were to interpret what this expression in equation 4.27 is counting we would first need to rearrange the terms in the summation so that adjacencies multiply the sum of two degrees

$$a_{su}(k_t + k_v) + a_{sv}(k_t + k_u) + a_{tu}(k_s + k_v) + a_{tv}(k_s + k_u)$$

before we could realise that, whenever one of the adjacencies a_{su}, a_{sv}, a_{tu} or a_{tv} equals 1, the summation in equation 4.27 adds the degree of the first and last vertices of the $\mathcal{L}_4$ induced by the edges st,uv and the adjacencies that equal 1. Therefore, it is easy to see that equation 4.27 is equivalent to

$$\frac{1}{2}\sum_{s\in V}\sum_{t\in\Gamma(s)}\sum_{u\in\Gamma(t)\setminus\{s\}}\sum_{v\in\Gamma(u)\setminus\{s,t\}}(k_s + k_v).$$

Now follows a series of basic algebraic transformations to obtain the same expression as in 4.28.

$$\begin{aligned}
&\frac{1}{2}\sum_{s\in V}\sum_{t\in\Gamma(s)}\sum_{u\in\Gamma(t)\setminus\{s\}}(k_s(k_u - 1 - a_{us}) + \xi(u) - a_{us}k_s - k_t)\\
&= \frac{1}{2}\sum_{s\in V}\sum_{t\in\Gamma(s)}\left(k_s \cdot \sum_{u\in\Gamma(t)\setminus\{s\}}(k_u - 1 - a_{us}) + \sum_{u\in\Gamma(t)\setminus\{s\}}\xi(u) - k_s \cdot \sum_{u\in\Gamma(t)\setminus\{s\}}a_{us} - k_t \cdot \sum_{u\in\Gamma(t)\setminus\{s\}}1\right)\\
&= \frac{1}{2}\sum_{s\in V}\sum_{t\in\Gamma(s)}\left(k_s(\xi(t) - k_s - k_t + 1 - |c(s,t)|) + \sum_{u\in\Gamma(t)\setminus\{s\}}\xi(u) - k_s|c(s,t)| - k_t(k_t - 1)\right).
\end{aligned}$$

The summation that remains inside the parenthesis in the last expression above had its inner summand changed from $\xi(u)$ to $\xi(s)$. This is correct since s and u are the endpoints of a $\mathcal{L}_3 = (s,t,u)$, and, in this case, it does not matter which endpoint's ξ we add. Put differently, this is true because s and u belong to the same orbit of $\mathcal{L}_3$. It only remains to state that

$$\sum_{u\in\Gamma(t)\setminus\{s\}}\xi(u) = (k_t - 1)\xi(s).$$

Expression in 4.28 can now be obtained through easy algebraic manipulations. □

We now give the last result that is used to devise an algorithm for an efficient way of computing $\mathbb{V}_{rla}[C_G]$ by evaluating equation 4.17.

Proposition 4.13. *Let $G = (V, E)$ be a graph and $Q = Q(G)$.*

$$\sum_{\{st,uv\}\in Q}(a_{su} + a_{sv} + a_{tu} + a_{tv})(k_s + k_t + k_u + k_v) = \sum_{s\in V}k_s \cdot N(s) \tag{4.29}$$

where $N(s)$ is defined as

$$N(s) = k_s(\xi(s) + 2 - 2k_s) + \sum_{t\in\Gamma(s)}(\xi(t) - 2k_t - 2|c(s,t)|). \tag{4.30}$$

Proof. Similarly as in proposition 4.12, we can see that the summation in equation 4.29 adds the degrees of the vertices of each $\mathcal{L}_4$ in G. Certainly, we could even follow the same strategy. However, unlike in that proposition, the degrees added are not only those of the endpoints, but also of the interior vertices. An equivalent expression of the left hand side of equation 4.29 is

$$\frac{1}{2}\sum_{s\in V}\sum_{t\in\Gamma(s)}\sum_{u\in\Gamma(t)\setminus\{s\}}\sum_{v\in\Gamma(u)\setminus\{s,t\}}(k_s + k_t + k_u + k_v).$$

Trying to simplify this expression is, in our opinion, hopeless. Nevertheless, it provides a good starting point to obtain the right hand side of equation 4.29. First, notice that the degree of a vertex, say, $w \in V$, will appear in the inner-most summation as many times as s can be found in paths of 4 vertices. Therefore, we only need to introduce $N(w)$, the number of $\mathcal{L}_4$ that a vertex w is part of. For this we can define the following procedure to compute all values of $N(w)$:

$N \leftarrow 0^n$
for $s \in V$, $t \in \Gamma(s)$, $u \in \Gamma(t) \setminus \{s\}$, $v \in \Gamma(u) \setminus \{s, t\}$ **do**
  $N_s \leftarrow N_s + 1$
  $N_t \leftarrow N_t + 1$
end for

The *for* loop in this procedure is actually a contraction of four *for* loops, one for each vertex s, t, u, and v, in this order. Notice that there is no need to divide by two at the end since s and t belong to different orbits (in other words, we would need the factor $1/2$ at the end if we also incremented positions N_u and N_v). We can do without the inner-most loop (see the procedure at the left), and from there removing the new inner-most loop (the loop for u) is easy (see the procedure at the right).

$N \leftarrow 0^n$
for $s \in V$, $t \in \Gamma(s)$, $u \in \Gamma(t) \setminus \{s\}$ **do**
  $N_s \leftarrow N_s + k_u - 1 - a_{su}$
  $N_t \leftarrow N_t + k_u - 1 - a_{su}$
end for

$N \leftarrow 0^n$
for $s \in V$, $t \in \Gamma(s)$ **do**
  $N_s \leftarrow N_s + \xi(t) - k_s - k_t + 1 - |c(s,t)|$
  $N_t \leftarrow N_t + \xi(t) - k_s - k_t + 1 - |c(s,t)|$
end for

Since we know that

$$\sum_{s \in V} \sum_{t \in \Gamma(s)} f(s,t) = \sum_{st \in E} (f(s,t) + f(t,s))$$

then the procedure can be further simplified into the procedure to the left. Then, for every vertex s, its degree $(-2k_s)$, the constant 2 and $\xi(s)$ are all added k_s times. Therefore, we can further simplify the procedure, and we obtain the one to the right.

$N \leftarrow 0^n$
for $st \in E$ **do**
  $N_s \leftarrow N_s + \xi(s) + \xi(t) - 2(k_s + k_t + 1 - |c(s,t)|)$
  $N_t \leftarrow N_t + \xi(s) + \xi(t) - 2(k_s + k_t + 1 - |c(s,t)|)$
end for

$N_s \leftarrow k_s(2 + \xi(s) - 2k_s)$, for all $s \in V$
for $st \in E$ **do**
  $N_s \leftarrow N_s + \xi(t) - 2k_t - 2|c(s,t)|$
  $N_t \leftarrow N_t + \xi(s) - 2k_s - 2|c(s,t)|$
end for

Now we can undo the contraction of the *for* loop and we obtain a procedure that computes $N(s)$ in the same way equation 4.30 does

$N_s \leftarrow k_s(2 + \xi(s) - 2k_s)$, for all $s \in V$
for $s \in V$ **do**
  for $t \in \Gamma(s)$ **do**
    $N_s \leftarrow N_s + \xi(t) - 2k_t - 2|c(s,t)|$
  end for
end for

□

4.3.2 Computing the variance in general graphs

The previous expressions for graph counting (and related) lay the foundations upon which we devise an algorithm to compute the variance of the number of crossings in general graphs, namely $\mathbb{V}_{rla}[C_G]$. Now, notice that if we were to evaluate all $n_G(\mathcal{L}_4)$, $n_G(\mathcal{L}_5)$, $n_G(\mathcal{C}_4)$, ..., at the same time we would easily see that quite a large amount of computations can be reused by storing them in memory. These are the results regarding the amount of common neighbours $|c(s,t)|$ of two vertices $s, t \in V$, which need not be adjacent and the sum of the degrees of all the common neighbours between two vertices. The algorithm is presented in pseudocode 4.2. We highlighted in red the parts that can be reused.

The algorithm uses several variables which, due to lack of space, we list here:

$$
\begin{aligned}
&L_{4,c_1} \leftarrow 0, L_{4,c_2} \leftarrow 0 && \text{// These refer to the second and third summations of equation 4.25}\\
&L_5 \leftarrow 0 && \text{// Equal to equation 4.26}\\
&L_G \leftarrow 0 && \text{// Equal to } \sum_{st\in E} k_s k_t\\
&\varphi \leftarrow 0, \epsilon \leftarrow 0, n_C \leftarrow 0, n_Z \leftarrow 0 && \text{// Equal to equations 4.18, 4.19, 4.22, 4.23}\\
&n_Y \leftarrow 0, \theta \leftarrow 0, \phi \leftarrow 0 && \text{// Equal to equations 4.24, 4.27, 4.29}\\
&N \leftarrow 0^n && \text{// } N(s), \text{ as in equation 4.30}
\end{aligned}
\tag{4.31}
$$

The most basic algorithm directly computes the results presented above, i.e., by only reusing the computations that are readily available, by evaluating the expressions above given an edge and accumulating the result in the appropriate variables. These computations are found within the main loop of the algorithm (in line 6) which iterates over the set of edges of the graph. The only computation that is reused in that pseudocode is the number of common vertices of two adjacent vertices. This can be found in line 12 where we declare a variable $c_{s,t}$ that is used to store the value $|c(s,t)|$ for the edge $\{s,t\} \in E$ that is being processed in the corresponding iteration of the main loop of line 6.

However, there are other computations which we can reuse but not so readily. These are marked in red. Although briefly, this was mentioned before: we are talking about the cardinality of the intersection of the neighbourhoods of two vertices that we do not know, a priori, whether they are adjacent or not, and the sum of the degrees of all the vertices in said intersection. Notice that the loops in lines 7 and 9 compute the amount of common neighbours between vertices t and $u_1 \in \Gamma(s) \setminus \{t\}$, and vertices s and $u_2 \in \Gamma(t) \setminus \{s\}$. Although we may not know whether t and u_1, or s and u_2, are connected, they might actually be, hence making the effort of storing computations be worthwhile since these will be reused in coming iterations of the main loop (line 6).

Proposition 4.14 analyses the algorithm when it does not reuse anything, and proposition 4.15 analyses the algorithm when it does. Perhaps not so surprisingly, the time complexity does not change while the space complexity increases. Even though, the second version of the algorithm seems to be faster than the first (see table 4.1).

Proposition 4.14. *Let $G = (V, E)$ be a graph. Assume that the graph is implemented using adjacency lists and that are sorted, namely, the adjacency list of vertex u, $\Gamma(u)$, contains labels that are sorted in increasing lexicographic order. Computing $\mathbb{V}_{rla}[C_G]$ with algorithm in pseudocode 4.2 has time complexity $O\left(k_{max} n \langle k^2 \rangle\right)$. The space complexity is $O(n)$.*

Proof. We need a linear amount of space in n to store the values of the function $\xi(s)$ and for $N(s)$ (from equation 4.30) for each vertex $s \in V$.

The cost of the intersection of two sorted adjacency lists $\Gamma(u)$ and $\Gamma(v)$ has cost the maximum degree of the two vertices $\Delta(u,v) = O(\max\{k_u, k_v\})$. Now, for each edge $\{s,t\} \in E$ the algorithm performs two intersection operations to compute the values $|c(t,u_1)|$ and $|c(s,u_2)|$, and also to compute the sum of the degrees of the vertices in that intersection. Since the second operation is done in constant time, the algorithm has cost

$$
\begin{aligned}
\sum_{st\in E} \left(\sum_{u_1 \in \Gamma(t)} \Delta(t,u_1) + \sum_{u_2 \in \Gamma(s)} \Delta(s,u_2) \right) &\leq k_{max} \sum_{st\in E} (k_s + k_t - 2)\\
&= k_{max}(n\langle k^2 \rangle - 2m) = O\left(k_{max} n \langle k^2 \rangle\right)
\end{aligned}
$$

since

$$
\sum_{st\in E} (k_s + k_t) = \sum_{u\in V} k_u^2 = n\langle k^2 \rangle.
$$

Notice that the assumption that the graph's adjacency list is sorted merely simplifies the algorithm. In case it was not, sorting it has cost $\sum_{u\in V} k_u \log k_u \leq \sum_{u\in V} k_u^2$. □

It is easy to see that the time complexity of algorithm in pseudocode 4.2 is bounded below by $\Omega(k_{max}m)$ and bounded above by $O\left(k_{max}m^2\right)$.

Improving the algorithm by reusing computations Reusing computations is rather easy. If we were to do so, we could make use of a hash table H to store the reusable computations where its keys are unordered pairs of vertices x and y that are adjacent ($\{x,y\} \in E$) or such that there exists another vertex z adjacent to both of them, i.e., $a_{xz} = a_{zy} = 1$. In the latter case we have a path of 3 vertices (x,z,y). Note that these cases are not mutually exclusive, and if two vertices are both adjacent and connected via a third vertex we only store it once. Recall that the reusable computations are, for a pair of two vertices $\{x,y\}$, the amount of common neighbours $|c(x,y)|$, and the sum of the degrees of the vertices that are neighbours of both x and y: $S_{x,y} = \sum_{u\in\Gamma(x)\cap\Gamma(y)} k_u$. Therefore, the algorithm would use an amount of extra space that is proportional to the amount of pairs of such vertices, which represents the size of H. In proposition 4.15 we study the complexity of algorithm 4.2 when reusing computations and we explain how to do it. See pseudocode 4.3 for details.

Proposition 4.15. *Let $G = (V,E)$ be a graph. Assume the graph to be as in proposition 4.14. When algorithm in pseudocode 4.2 reuses computations (see pseudocode 4.3) has as space complexity a function g such that*

$$g \in O\left(n + \min\left\{\binom{n}{2}, m + n_G(\mathcal{L}_3)\right\}\right). \tag{4.32}$$

The time complexity, however, remains the same.

Proof. The description of the conditions in which the algorithm stores, in the hash table H, pairs of vertices speaks for itself. Sparser graphs may contain few cycles of 3 vertices therefore many endpoints of 3-paths are not connected and should be counted as unique pairs. Denser graphs have more cycles, so the connections between these endpoints are more likely to be adjacent hence making an edge. Then, the summation $O(m + n_G(\mathcal{L}_3))$ may count several edges twice. The denser the graph the more likely this is to happen. That is why we take the minimum between all possible pairs $\binom{n}{2}$ and $m + n_G(\mathcal{L}_3)$. The term n is added to take into account the memory used to store the functions $\xi(s)$ and $N(s)$ (the latter is defined in equation 4.30).

The algorithm in pseudocode 4.2 can be modified very easily. Whenever the algorithm needs one of these two values, first find $\{x,y\}$ in H. If H has such pair, use the appropriate values associated to it. If not, the new algorithm must compute both $|c(x,y)|$ and $S_{x,y} = \sum_{u\in\Gamma(x)\cap\Gamma(y)} k_u$. which can be computed simultaneously and in time $O(\max\{k_x,k_y\})$. Assuming that H has constant time complexity in look-ups and insertions, the time complexity does not change. □

Performance of the two algorithms In practice the algorithm that reuses computations is more efficient than the algorithm that does not, specially in dense graphs. Table 4.1 shows the speed up obtained when the algorithm reuses computations. These are the quotient of the execution time between the algorithm that does not reuse computations and the algorithm that does. The execution times were measured on random graphs $G_{n,p} \in \mathcal{G}_{n,p}$ for several values of n and p. For each pair of values we generated 10 graphs and executed the algorithm an amount of times that depended on p – the smaller the probability the more times the algorithm was executed.

It is interesting to observe that the speed up not only increases with the density of the graph – for a fixed n, a higher amount of edges requires more computations so reusing them saves time–, but it also increases with the value of n – for a fixed p the speed up increases with increasing values of n. The latter claim, however, seems to be false for a fixed $p < 0.05$ and increasing values of n, but this should

Algorithm 4.2: Algorithm to calculate $\mathbb{V}_{rla}[C_G]$ in general graphs in time $O\left(k_{max} n \langle k^2 \rangle\right)$.

Input: $G = (V, E)$ a graph as described in proposition 4.14.
Output: $\mathbb{V}_{rla}[C_G]$, the variance of the number of crossings.

Function VARIANCEC(G) **is**
  // Declare variables listed in 4.31
  $\xi_s \leftarrow 0^n$
  for $s \in V$ **do**
    $\xi_s \leftarrow \xi(s)$
    $N_s \leftarrow k_s(\xi_s + 2 - 2k_s)$
  for $\{s, t\} \in E$ **do**
    for $u_1 \in \Gamma(s) \setminus \{t\}$ **do**
      $L_5 \leftarrow L_5 - |c(t, u_1)| + (k_t - 1 - a_{tu_1})(k_u - 1 - a_{tu_1}) + 1$
    for $u_2 \in \Gamma(t) \setminus \{s\}$ **do**
      $L_5 \leftarrow L_5 - |c(s, u_2)| + (k_s - 1 - a_{su_2})(k_u - 1 - a_{su_2}) + 1$
      $n_C \leftarrow n_C + |c(s, u_2)| - 1$
    $c_{s,t} \leftarrow 0$ // Equal to $|c(s, t)|$
    for $u \in \Gamma(t) \cap \Gamma(s)$ **do**
      $c_{s,t} \leftarrow c_{s,t} + 1$
      $n_Z \leftarrow n_Z + k_u$
      $n_Y \leftarrow n_Y - k_u$
    $L_G \leftarrow L_G + k_s k_t$
    $n_Z \leftarrow n_Z - 2c_{s,t}$
    $n_Y \leftarrow n_Y + (m - k_s - k_t + 3)c_{s,t}$
    $\varphi \leftarrow \varphi + k_s k_t (m - k_s - k_t + 1)$
    $\epsilon \leftarrow \epsilon + (k_s + k_t)(n\langle k^2 \rangle - \xi_s - \xi_t - k_t(k_t - 1) - k_s(k_s - 1))$
    $\theta \leftarrow \theta + k_s(\xi_t - k_s - k_t + 1) + (k_t - 1)(\xi_s - k_t)$
    $\theta \leftarrow \theta + k_t(\xi_s - k_s - k_t + 1) + (k_s - 1)(\xi_t - k_s)$
    $\theta \leftarrow \theta - 2(k_s + k_t)c_{s,t}$
    $L_{4,c_1} \leftarrow L_{4,c_1} + \xi_s + \xi_t$
    $L_{4,c_2} \leftarrow L_{4,c_2} + c_{s,t}$
    $N_s \leftarrow N_s + \xi_t - 2(k_t + c_{s,t})$
    $N_t \leftarrow N_t + \xi_s - 2(k_s + c_{s,t})$
  for $s \in V$ **do**
    $\phi \leftarrow \phi + k_s N_s$
  $|Q| \leftarrow (m(m+1) - n\langle k^2 \rangle)/2$ // see [7]
  $K_G \leftarrow (m+1)n\langle k^2 \rangle - n\langle k^3 \rangle - 2L_G$ // see [2, Proposition 4.4]
  $\epsilon \leftarrow \epsilon/2$
  $L_4 \leftarrow m - n\langle k^2 \rangle + L_{4,c_1}/2 - L_{4,c_2}$
  $L_5 \leftarrow L_5/2$
  $n_C \leftarrow n_C/4$
  $n_Y \leftarrow n_Y/3$
  $\theta \leftarrow \theta/2$
  // Compute the variance
  $V \leftarrow \frac{2}{45}(m+2)|Q| - \frac{1}{180}L_5 - \frac{2m+7}{180}L_4 - \frac{3}{45}n_C + \frac{1}{90}K_G$
  $V \leftarrow V + \frac{1}{30}n_Y + \frac{1}{30}n_Z$
  $V \leftarrow V - \frac{1}{60}\theta + \frac{1}{180}\phi + \frac{1}{180}\epsilon - \frac{1}{90}\varphi$
  return V

Algorithm 4.3: Algorithm to calculate $\mathbb{V}_{rla}[C_G]$ in general graphs in time $O\left(k_{max} n \langle k^2 \rangle\right)$ reusing computations.

Input: $G = (V, E)$ a graph as described in proposition 4.14, a hash table H indexed with keys pairs of vertices, and two vertices $x, y \in V$.
Output: The values $|c(x, y)|$ and $S_{x,y} = \sum_{u \in \Gamma(x) \cap \Gamma(y)} k_u$.

Function ComputeAndStore(H, G, x, y) **is**
- $c_{x,y} \leftarrow 0$ // Equal to $|c(x, y)|$.
- $S_{x,y} \leftarrow 0$ // Equal to $\sum_{u \in \Gamma(x) \cap \Gamma(y)} k_u$.
- **if** $\{x, y\} \notin H$ **then**
  - **for** $u \in \Gamma(x) \cap \Gamma(y)$ **do**
    - $c_{x,y} \leftarrow c_{x,y} + 1$
    - $S_{x,y} \leftarrow S_{x,y} + k_u$
  - // Store $c_{x,y}$ and $S_{x,y}$ in the hash table indexed with key $\{x, y\}$.
  - $H \leftarrow H \cup \langle \{x, y\}, c_{x,y}, S_{x,y} \rangle$
- **else**
  - $c_{x,y} \leftarrow H(\{x, y\}).c_{x,y}$ // Retrieve the values from H.
  - $S_{x,y} \leftarrow H(\{x, y\}).S_{x,y}$
- **return** $c_{x,y}$, $S_{x,y}$

Input: $G = (V, E)$ a graph as described in proposition 4.14.
Output: $\mathbb{V}_{rla}[C_G]$, the variance of the number of crossings.

Function VarianceC(G) **is**
- // Include lines from 1 to 5 of algorithm 4.2.
- $H \leftarrow \emptyset$ // Initialise hash table.
- **for** $\{s, t\} \in E$ **do**
  - **for** $u_1 \in \Gamma(s) \setminus \{t\}$ **do**
    - $c(t, u_1) \leftarrow 0$ // Equal to $|c(t, u_1)|$.
    - // Compute $|c(t, u_1)|$ and $|S_{t,u_1}|$ and store them in H,
    - // or retrieve $|c(t, u_1)|$ from H.
    - $c_{t,u_1}, \_ \leftarrow$ ComputeAndStore(G, H, t, u_1)
    - $L_5 \leftarrow L_5 - c_{t,u_1} + (k_t - 1 - a_{tu_1})(k_u - 1 - a_{tu_1}) + 1$
  - **for** $u_2 \in \Gamma(t) \setminus \{s\}$ **do**
    - $c(s, u_2) \leftarrow 0$ // Equal to $|c(s, u_2)|$.
    - // Compute $|c(s, u_2)|$ and $|S_{s,u_2}|$ and store them in H,
    - // or retrieve $|c(s, u_2)|$ from H.
    - $c_{s,u_2}, \_ \leftarrow$ ComputeAndStore(G, H, s, u_2)
    - $L_5 \leftarrow L_5 - c_{s,u_2} + (k_s - 1 - a_{su_2})(k_u - 1 - a_{su_2}) + 1$
    - $n_C \leftarrow n_C + c_{s,u_2} - 1$
  - $c_{s,t} \leftarrow 0$ // Equal to $|c(s, t)|$.
  - $S_{s,t} \leftarrow 0$ // Equal to $\sum_{u \in \Gamma(s) \cap \Gamma(t)} k_u$.
  - // Compute values and store them in H or retrieve them from H.
  - $c_{s,t}, S_{s,t} \leftarrow$ ComputeAndStore(G, H, s, t)
  - $n_Z \leftarrow n_Z + S_{s,t}$
  - $n_Y \leftarrow n_Y - S_{s,t}$
  - // Include lines from 17 to 28 of algorithm 4.2.
  - // Recall some of them use the value $c_{s,t}$.
- // Include lines from 28 to 42 of algorithm 4.2.
- **return** V

not be regarded as a downside as these graphs are very sparse, namely the algorithm that does not reuse computations would perform well enough on them since the common neighbours of non-adjacent nodes are scarce.

It is worth highlighting that while reusing computations can make the algorithm 6 times faster (see the speed up for $n = 150$, $p = 0.70$), it can also make it 3 times slower (for $n = 150$, $p \leq 0.03$). This is due to the overhead of storing computations that might never be reused. There is no reason to think that the speed up stops decreasing with increasing values of n for a fixed $p < 0.05$, or that it stops increasing for increasing values of n for a fixed $p \geq 0.10$. Therefore, a practical implementation of the computation of $\mathbb{V}_{rla}[C_G]$ should combine both implementations.

$n \setminus p$	0.01	0.02	0.03	0.04	0.05	0.10	0.15	0.20
10	1.009	1.031	1.002	0.931	0.935	0.936	0.911	0.757
20	0.963	0.964	0.886	0.873	0.842	0.645	0.589	0.632
30	0.964	0.934	0.783	0.689	0.629	0.510	0.556	0.668
40	0.894	0.756	0.626	0.544	0.504	0.498	0.611	0.807
50	0.810	0.643	0.512	0.467	0.436	0.501	0.689	0.938
60	0.769	0.548	0.442	0.416	0.409	0.543	0.755	1.063
70	0.614	0.433	0.402	0.371	0.382	0.544	0.799	1.140
80	0.565	0.391	0.357	0.369	0.380	0.598	0.870	1.276
90	0.507	0.367	0.352	0.366	0.391	0.637	0.964	1.363
100	0.446	0.354	0.351	0.370	0.411	0.671	1.038	1.461
110	0.399	0.328	0.338	0.366	0.401	0.687	1.071	1.505
120	0.380	0.329	0.343	0.391	0.431	0.737	1.156	1.624
130	0.367	0.326	0.348	0.390	0.441	0.782	1.211	1.703
140	0.347	0.322	0.361	0.405	0.457	0.812	1.278	1.786
150	0.330	0.331	0.359	0.415	0.480	0.844	1.331	1.875

$n \setminus p$	0.30	0.40	0.50	0.60	0.70	0.80	0.90	1.00
10	0.793	0.747	0.865	0.973	0.951	1.173	1.232	1.198
20	0.813	0.989	1.284	1.541	1.723	1.763	1.691	1.452
30	1.014	1.402	1.793	2.070	2.229	2.166	1.938	1.498
40	1.253	1.736	2.168	2.495	2.590	2.493	2.188	1.592
50	1.474	2.027	2.512	2.791	2.930	2.809	2.432	1.659
60	1.672	2.270	2.758	3.112	3.284	3.127	2.627	1.754
70	1.832	2.468	3.031	3.474	3.630	3.435	2.803	1.721
80	2.024	2.680	3.327	3.757	3.942	3.702	3.051	1.834
90	2.181	2.913	3.596	4.089	4.239	4.042	3.288	1.937
100	2.323	3.107	3.824	4.350	4.557	4.368	3.546	1.999
110	2.372	3.197	3.945	4.497	4.834	4.624	3.727	2.118
120	2.565	3.457	4.269	4.911	5.159	4.943	3.950	2.067
130	2.700	3.669	4.530	5.237	5.517	5.183	3.997	2.179
140	2.838	3.855	4.763	5.507	5.787	5.428	4.285	2.179
150	2.969	4.024	5.055	5.850	6.085	5.596	4.498	2.123

Table 4.1: Speed up of algorithm 4.2 when it reuses computations, measured as the quotient between the execution time of the algorithm when it does not reuse computations and the execution time of the same algorithm when it does. For each pair of n and p we generated 10 random graphs $G_{n,p}$. Then we executed the algorithm $k+1$ times, but discarded the first and averaged the rest. We used different values of k depending on the value of p: for $p < 0.05$ we used $k = 1000$, for $0.05 \leq p \leq 0.15$ we used $k = 100$, and for $p \geq 0.2$ we used $k = 10$.

4.3.3 The case of trees

Computing the variance of C on trees, namely $\mathbb{V}_{rla}[C_T]$ is simpler and the time and space complexities can be both reduced to $O(n)$. This algorithm's pseudocode can be found in 4.4.

This complexity is achieved by taking notice of a few simple facts that hold only on trees, gathered in propositions 4.16, 4.17, 4.18, and 4.19. These results are obtained from scratch, i.e., we did not instantiate the previous (see 4.10, 4.11, 4.12 and 4.13) even though they could have been by taking note that, in trees, two adjacent vertices s and t do not have vertices in common, namely $|c(s,t)| = 0$. Also, we did not instantiate Movarraei's expressions for the counting of $\mathcal{L}_4$ and $\mathcal{L}_5$ given in [23] since it is not trivial at all, and our new approach is much simpler.

The first two propositions, which give a simple formulation for counting paths of 4 and 5 vertices respectively in trees, do not need a highly detailed proof due to their simplicity.

Proposition 4.16. *Let $T = (V, E)$ be a tree.*

$$n_T(\mathcal{L}_4) = \sum_{st \in E} (k_s - 1)(k_t - 1). \tag{4.33}$$

Proof. The product of the degrees of two adjacent vertices s and t (each of them minus 1) gives the amount of $\mathcal{L}_4$ with centroids s and t since the vertices do not share common neighbours. This is true in trees. The $\mathcal{L}_4$ with centroids s and t are only counted once. Finally, by adding up all these products for all edges in the tree we obtain the amount of $\mathcal{L}_4$ in the tree. □

Proposition 4.17. *Let $T = (V, E)$ be a tree.*

$$n_T(\mathcal{L}_5) = \frac{1}{2} \sum_{s \in V} \sum_{\substack{t \in V: \\ ts \in E}} (k_t - 1)\left(\xi(s) - k_t - k_s + 1\right). \tag{4.34}$$

Proof. We only need to realise the simple fact that:

$$n_T(\mathcal{L}_5) = \sum_{s\in V}\left(\frac{1}{2}\sum_{t\in\Gamma(s)}\sum_{u\in\Gamma(s)\setminus\{t\}}(k_t-1)(k_u-1)\right). \tag{4.35}$$

The proof is similar to proposition 4.16's proof. Any $\mathcal{L}_5$ has only centroid. Let $s\in V$ be such centroid. For any pair of different neighbours of s, $t\in\Gamma(s)$ and $u\in\Gamma(s)\setminus\{t\}$, the product $(k_t-1)(k_u-1)$ gives the amount of $\mathcal{L}_5$ with centroid s and through vertices t and u. The two inner summations of equation 4.35 count such paths, twice. It simply remains to keep applying algebraic transformations to obtain the desired expression. □

Notice that these two expressions could have been derived from the general formulae by Movarraei (see [23], equations 4.20 and 4.21), but we would have reached the same conclusions through a seemingly much longer path.

The next two propositions simplify terms that are, at a first glance, difficult to evaluate. These simplifications can be used to reduce the time complexity of their computation at the expense of $O(n)$-space complexity. These two, like the previous two propositions, only hold on trees.

Proposition 4.18. *Let $T=(V,E)$ be a tree and $Q=Q(T)$. The expression*

$$\sum_{\{st,uv\}\in Q}(a_{su}+a_{sv}+a_{tu}+a_{tv})(k_s+k_t+k_u+k_v) \tag{4.36}$$

is equal to:

$$\sum_{st\in E}\left((k_s-1)(k_t-1)(k_s+k_t)+(k_t-1)(\xi(s)-k_t)+(k_s-1)(\xi(t)-k_s)\right). \tag{4.37}$$

Proof. In [2] it was proved that, given a fixed $\{st,uv\}\in Q(G)$, where G is any simple graph, the sum $a_{su}+a_{sv}+a_{tu}+a_{tv}$ counts the amount of $\mathcal{L}_4$ with vertices s,t,u,v in G (see [2, Proposition 4.1]). Since we are dealing with trees, we have that $a_{su}+a_{sv}+a_{tu}+a_{tv}\leq 1$. Therefore, in equation 4.36, the degrees of the vertices that make up the $\mathcal{L}_4$ in the tree are added up. Therefore

$$\sum_{\{st,uv\}\in Q}(a_{su}+a_{sv}+a_{tu}+a_{tv})(k_s+k_t+k_u+k_v)=\sum_{\mathcal{L}_4=(s,t,u,v)\in T}(k_s+k_t+k_u+k_v).$$

Obtaining equation 4.37 is obtained with simple algebraic manipulations

$$\begin{aligned}\sum_{\mathcal{L}_4=(s,t,u,v)\in T}(k_s+k_t+k_u+k_v) &= \sum_{st\in E}\sum_{u\in\Gamma(s)\setminus\{t\}}\sum_{v\in\Gamma(t)\setminus\{s\}}(k_s+k_t+k_u+k_v)\\ &= \sum_{st\in E}\left((k_s+k_t)(k_s-1)(k_t-1)+\sum_{u\in\Gamma(s)\setminus\{t\}}\sum_{v\in\Gamma(t)\setminus\{s\}}(k_u+k_v)\right).\end{aligned}$$

It only remains to show that, for any edge $\{s,t\}\in E$,

$$\begin{aligned}\sum_{u\in\Gamma(s)\setminus\{t\}}\sum_{v\in\Gamma(t)\setminus\{s\}}(k_u+k_v) &= \sum_{u\in\Gamma(s)\setminus\{t\}}\sum_{v\in\Gamma(t)\setminus\{s\}}k_u+\sum_{u\in\Gamma(s)\setminus\{t\}}\sum_{v\in\Gamma(t)\setminus\{s\}}k_v\\ &= \sum_{u\in\Gamma(s)\setminus\{t\}}k_u(k_t-1)+\sum_{u\in\Gamma(s)\setminus\{t\}}(\xi(t)-k_s)\\ &= (k_t-1)(\xi(s)-k_t)+(k_s-1)(\xi(t)-k_s).\end{aligned}$$

Hence equation 4.37. □

Proposition 4.19. *Let $T = (V, E)$ be a tree and $Q = Q(T)$. The expression*

$$\sum_{\{st,uv\}\in Q} (k_s(a_{tu} + a_{tv}) + k_t(a_{su} + a_{sv}) + k_u(a_{sv} + a_{tv}) + k_v(a_{su} + a_{tu})) \tag{4.38}$$

is equal to:

$$\sum_{st\in E} ((k_t - 1)(\xi(s) - k_t) + (k_s - 1)(\xi(t) - k_s)) . \tag{4.39}$$

Proof. The proof is similar to the proof of proposition 4.18. The first step is to rearrange the terms in equation 4.38:

$$\sum_{\{st,uv\}\in Q} (a_{tu}(k_s + k_v) + a_{tv}(k_s + k_u) + a_{su}(k_t + k_v) + a_{sv}(k_t + k_u)) .$$

Again, in trees $a_{su} + a_{sv} + a_{tu} + a_{tv} \leq 1$. Therefore, since one of these adjacencies being 1 produces a $\mathcal{L}_4$, whichever adjacency it is that equals one, this expressions adds up the degrees of the leaves of this $\mathcal{L}_4$. Therefore, equation 4.38 is equal to

$$\sum_{st\in E} \sum_{u\in\Gamma(s)\setminus\{t\}} \sum_{v\in\Gamma(t)\setminus\{s\}} (k_u + k_v).$$

The derivation of this expression is already done in the proof of proposition 4.18. □

As mentioned at the beginning of section 4.3, although propositions 4.5 and 4.6 are results for general graphs they also prove to be useful for the computation of the variance in trees without the need of further simplification.

The algorithm for calculating $\mathbb{V}_{rla}[C_T]$ is presented in 4.4. As the reader will see, it assumes a labelling of the vertices from 1 to n and that it is implemented using adjacency lists – in which the two vertices of every edge can be found in the adjacency list of their respective neighbour. When referring to a vertex $s \in V$ we assume that s is also a numerical value in $[n]$. Therefore, if s is a vertex in the graph, and t is one of its neighbours, we use the comparison $s < t$ to indicate that the numerical value of the label of s is smaller than the numerical value of the label of t. The algorithm makes use of the results presented in this section (see 4.5, 4.6, 4.16, 4.17, 4.18, 4.19), to obtain a time complexity of $O(n)$.

Algorithm 4.4: Computing $\mathbb{V}_{rla}[C_T]$ in time and space $O(n)$.

Input: $T = (V, E)$ a tree.
Output: $\mathbb{V}_{rla}[C_T]$, the variance of the number of crossings in T.

1 **Function** VarianceC(T) **is**
2 $\quad \xi_s \leftarrow 0^n$
3 $\quad$ **for** $s \in V$ **do**
4 $\quad\quad \xi_s \leftarrow \xi(s)$
5 $\quad L_4 \leftarrow 0,\ L_5 \leftarrow 0,\ L_G \leftarrow 0$
6 $\quad \varphi \leftarrow 0$ `// equal to equation 4.18`
7 $\quad \epsilon \leftarrow 0$ `// equal to equation 4.19`
8 $\quad \phi \leftarrow 0$ `// equal to equation 4.36`
9 $\quad \theta \leftarrow 0$ `// equal to equation 4.38`
10 $\quad$ **for** $\{s, t\} \in E$ **do**
11 $\quad\quad L_G \leftarrow L_G + k_s k_t$
12 $\quad\quad L_5 \leftarrow L_5 + (k_t - 1)(\xi_s - k_t - k_s + 1) + (k_s - 1)(\xi_t - k_t - k_s + 1)$
13 $\quad\quad L_4 \leftarrow L_4 + (k_s - 1)(k_t - 1)$
14 $\quad\quad \phi \leftarrow \phi + (k_s - 1)(k_t - 1)(k_s + k_t) + (k_t - 1)(\xi_s - k_t) + (k_s - 1)(\xi(t) - k_s)$
15 $\quad\quad \theta \leftarrow \theta + (k_t - 1)(\xi_s - k_t) + (k_s - 1)(\xi_t - k_s)$
16 $\quad\quad \epsilon \leftarrow \epsilon + (k_s + k_t)(n\langle k^2\rangle - \xi_s - \xi_t - k_t(k_t - 1) - k_s(k_s - 1))$
17 $\quad\quad \varphi \leftarrow \varphi + k_s k_t (m - k_s - k_t + 1)$
18 $\quad |Q| \leftarrow (n(n-1) - n\langle k^2\rangle)/2$ `// see [7]`
19 $\quad K_G \leftarrow (m+1)n\langle k^2\rangle - n\langle k^3\rangle - 2L_G$ `// see [2, Proposition 4.4]`
20 $\quad K_G \leftarrow (m+1)n\langle k^2\rangle - n\langle k^3\rangle - 2L_G$
21 $\quad L_5 \leftarrow L_5/2$
22 $\quad \epsilon \leftarrow \epsilon/2$
$\quad$ `// Compute the variance.`
23 $\quad V \leftarrow \frac{2}{45}(m+2)|Q| - \frac{1}{180}L_5 - \frac{2m+7}{180}L_4 + \frac{1}{90}K_G$
24 $\quad V \leftarrow V - \frac{1}{60}\theta + \frac{1}{180}\phi + \frac{1}{180}\epsilon - \frac{1}{90}\varphi$
25 $\quad$ **return** V

Proposition 4.20. *Let $T = (V, E)$ be a tree. Algorithm 4.4 computes $\mathbb{V}_{rla}[C_T]$ (see equation 4.13). It does so in time and space $O(n)$.*

Proof. First we show that the computation of the different terms involved in the formula for the variance of C in trees (see equation 4.13) is correct. We only need to pay attention to the computation of $n_T(\mathcal{L}_5)$. Its computation in line 12 is correct in spite of not implementing exactly equation 4.34. The only reason for this is the following equality:

$$\sum_{s \in V} \sum_{t \in \Gamma(s)} f(s,t) = \sum_{st \in E} (f(s,t) + f(t,s))$$

where $f(s,t)$ is a real-valued function on the vertices s and t. Equation 4.34 is computed this way so as to avoid traversing the set of edges E twice.

Therefore, the computation of $n_T(\mathcal{L}_5)$ is correct since we use the result in proposition 4.17 (line 12). The computation of $n_T(\mathcal{L}_4)$ is also correct since it applies the result in proposition 4.16 (line 13). The term K_G, the last two terms in equation 4.13, and the size of Q are computed using results presented in [2]. The results in propositions 4.18 and 4.19 give alternative expressions for equations 4.36 and 4.38 respectively. These expressions are used here correctly in lines 14 and 15. Finally, propositions 4.5 and 4.6 are applied correctly in lines 16 and 17.

In order to keep the time complexity to the minimum, we need to store the values $\xi(s)$ for all vertices $s \in V$. Therefore, the space complexity of the algorithm is $O(n)$. The time complexity is easy

to analyse: computing all values of ξ_s (line 3) takes $O(n)$ time and since we only iterate over the set of edges once then the time complexity is $O(m) = O(n)$. Note that the value $n\langle k^2\rangle$ can be stored in a single integer variable (hence constant space) and computed in $O(n)$ at the same time the array ξ_s is computed. □

4.3.4 The case of forests

The variance on forests can be computed straightforwardly using the algorithm for trees. Let $F = \{T_i\}_{i=1}^k$ be a forest of k trees. Assuming the connected components are readily available, a direct implementation computes the values of the variables declared in lines from 5 to 9 of algorithm 4.4, as the accumulation of the amount of $\mathcal{L}_4$ in every, the amount of $\mathcal{L}_5$ in every tree, and so on. The algorithm is given in pseudocode 4.5). In case the connected components were not readily available, finding them can be done using a breadth-first search, which has a time and space complexity of $O(n)$. The complexity of such algorithm is studied in proposition 4.21.

Algorithm 4.5: Computing $\mathbb{V}_{rla}[C_F]$ in time and space $O(n)$.

Input: $F = \{T_i\}_{i=1}^k$ a forest.
Output: $\mathbb{V}_{rla}[C_F]$, the variance of the number of crossings in F.

1 **Function** VarianceC(F) **is**
2 $\quad\xi_s \leftarrow 0^n$
3 $\quad$**for** $s \in V$ **do**
4 $\quad\quad\xi_s \leftarrow \xi(s)$
5 $\quad L_4 \leftarrow 0,\ L_5 \leftarrow 0,\ L_G \leftarrow 0$
6 $\quad\varphi \leftarrow 0$ `// equal to equation 4.18`
7 $\quad\epsilon \leftarrow 0$ `// equal to equation 4.19`
8 $\quad\phi \leftarrow 0$ `// equal to equation 4.36`
9 $\quad\theta \leftarrow 0$ `// equal to equation 4.38`
10 $\quad$**for** i *from* 1 *to* k **do**
11 $\quad\quad$**for** $\{s,t\} \in E(T_i)$ **do**
12 $\quad\quad\quad L_G \leftarrow L_G + k_s k_t$
13 $\quad\quad\quad L_5 \leftarrow L_5 + (k_t-1)(\xi_s - k_t - k_s + 1) + (k_s - 1)(\xi_t - k_t - k_s + 1)$
14 $\quad\quad\quad L_4 \leftarrow L_4 + (k_s - 1)(k_t - 1)$
15 $\quad\quad\quad \phi \leftarrow \phi + (k_s-1)(k_t-1)(k_s+k_t) + (k_t-1)(\xi_s - k_t) + (k_s-1)(\xi(t) - k_s)$
16 $\quad\quad\quad \theta \leftarrow \theta + (k_t-1)(\xi_s - k_t) + (k_s-1)(\xi_t - k_s)$
17 $\quad\quad\quad \epsilon \leftarrow \epsilon + (k_s+k_t)(n\langle k^2\rangle - \xi_s - \xi_t - k_t(k_t-1) - k_s(k_s-1))$
18 $\quad\quad\quad \varphi \leftarrow \varphi + k_s k_t (m - k_s - k_t + 1)$
19 $\quad|Q| \leftarrow (n(n-1) - n\langle k^2\rangle)/2$ `// see [7]`
20 $\quad K_G \leftarrow (m+1)n\langle k^2\rangle - n\langle k^3\rangle - 2L_G$ `// see [2, Proposition 4.4]`
21 $\quad L_5 \leftarrow L_5/2$
22 $\quad\epsilon \leftarrow \epsilon/2$
$\quad$`// Compute the variance.`
23 $\quad V \leftarrow \frac{2}{45}(m+2)|Q| - \frac{1}{180}L_5 - \frac{2m+7}{180}L_4 + \frac{1}{90}K_G$
24 $\quad V \leftarrow V - \frac{1}{60}\theta + \frac{1}{180}\phi + \frac{1}{180}\epsilon - \frac{1}{90}\varphi$
25 $\quad$**return** V

Proposition 4.21. *Let $F = \{T_i\}_{i=1}^k$ be a forest. Algorithm 4.5 computes $\mathbb{V}_{rla}[C_F]$. It does so in time and space $O(n)$.*

Proof. The correctness can be easily seen. Since there are no loops in F the terms of equation 4.17 that are not trivially 0 are those in the equation for the variance of C_G on trees (see equation 4.13).

These terms, the amount of $\mathcal{L}_4$, of $\mathcal{L}_5$, and the summations

$$
\begin{aligned}
S_1(F) &= \sum_{\{st,uv\}\in Q(F)} (k_s(a_{tu}+a_{tv})+k_t(a_{su}+a_{sv})+k_u(a_{sv}+a_{tv})+k_v(a_{su}+a_{tu})) \\
S_2(F) &= \sum_{\{st,uv\}\in Q(F)} (a_{su}+a_{sv}+a_{tu}+a_{tv})(k_s+k_t+k_u+k_v) \\
S_3(F) &= \sum_{\{st,uv\}\in Q(F)} (k_s+k_t)(k_u+k_v) \\
S_4(F) &= \sum_{\{st,uv\}\in Q(F)} (k_sk_t+k_uk_v),
\end{aligned}
$$

can be evaluated for every tree individually and then accumulate the results, i.e.

$$
n_F(\mathcal{L}_4)=\sum_{i=1}^{k} n_{T_i}(\mathcal{L}_4), \quad n_F(\mathcal{L}_5)=\sum_{i=1}^{k} n_{T_i}(\mathcal{L}_5), \quad S_j(F)=\sum_{i=1}^{k} S_j(T_i), \quad \forall 1\le j\le 4,
$$

since, by definition, a forest is the disjoint union of the $\{T_i\}_{i=1}^{k}$.

The complexity of the algorithm when the trees are readily available, namely when they need not be extracted using any method, is obviously $O(n)$ since loops in lines 10 and 11 iterate over the set of edges of F. It is easy to see that

$$
|E(F)|=\sum_{i=1}^{k}|E(T_i)|=\sum_{i=1}^{k}(|V(T_i)|-1)=|V(F)|-k=O(|V(F)|)=O(n).
$$

It is also obvious that the space complexity is $O(n)$.

Extracting the connected components, i.e., the T_i, can be done with a simple breadth-first search exploration of the forest. Start at any vertex $u\in V(F)$, apply the BFS on that vertex, store the vertices of the search in an array and mark them as visited. Then, find another vertex that has not been visited yet and repeat the same step. Repeat this process until vertices have been visited. The time needed to explore the vertices of the forests is $O(n)$ and the space complexity of the search is also $O(n)$. These extra costs do not affect the complexity of algorithm in pseudocode 4.5. □

5 Application to Syntactic Dependency Treebanks

We culminate this work by combining all our efforts into an application that is key for the verification of a long-posed hypothesis. This hypothesis predicts that the scarcity of dependency crossings in sentences is a side effect of dependency length minimisation (see [21] for a comprehensive discussion on the topic). In this section we study a weaker version of this hypothesis, which is the necessary condition of causality, namely we study whether there is a strong correlation between the number of crossings and the sum of the length of the dependencies. The contribution of this chapter is two-fold: firstly, we investigate the correlation between these two variables using a novel technique, and, secondly we investigate the nature of this correlation in the sense of how it arises. In order to do this, we analysed data from the Universal Dependencies 2.3 [24], Prague Dependencies [16], and Stanford Dependencies [20] datasets, henceforth briefly referred to as *UD 2.3*, *Prague* and *Stanford*. These datasets are made up of several treebanks, one for each language in the dataset, which in turn contain syntactic dependency trees. From a graph theoretical point of view, these syntactic dependency trees are labelled trees in which vertices represent words of a sentence and edges represent syntactic dependencies between pairs of words. Words can also be interpreted as the labels of the vertices. In order to avoid confusion, we distinguish between the syntactic dependency trees, referred to as "sentences", and the unlabelled tree of a sentence, referred to as "tree".

Providing evidence to support such hypothesis, however, is not so simple. Merely because the magnitudes D and C have different distributions so their relationship cannot be compared so easily (see figure 5.1(left) for a graph of D vs C). In order to do so, we need to standardise them, and we chose to use z-scores. More precisely, we standardised both D and C into D_z and C_z and analysed whether there is a strong correlation between these two. Standardisation is a widely applied method in statistics, an example of which can be found in [10, Figure 6] where, of two variables, only one of them is standardised. For any random variable X, its z-score standardisation is

$$X_z = \frac{X - \mathbb{E}[X]}{\sqrt{\mathbb{V}[X]}}. \tag{5.1}$$

We chose to standardise both D and C using z-scores with respect to random linear arrangements. Then,

$$D_z = \frac{D - \mathbb{E}_{rla}[D]}{\sqrt{\mathbb{V}_{rla}[D]}}, \tag{5.2}$$

$$C_z = \frac{C - \mathbb{E}_{rla}[C]}{\sqrt{\mathbb{V}_{rla}[C]}}. \tag{5.3}$$

This is where our work comes into play. Recall that with our algorithms we are now able to compute the exact value of the variance of C in uniformly random linear arrangements, namely $\mathbb{V}_{rla}[C]$, in all simple graphs (see pseudocode 4.3). Needless to say that without an algorithm for its exact computation we need to approximate it. Doing so, in any graph of n vertices, is quite likely to consist on running a Monte Carlo procedure that would have to execute quite a large number of iterations proportional to the amount of possible linear arrangements, $n!$, so as to keep the error of the measurements as low as possible, in which every one of them we would have to generate a linear arrangement π_r uniformly at random and compute $C_T(\pi_r)$. As we have discussed in section 2.4, the most efficient algorithm to do this in trees, to our best knowledge, is the stack-based algorithm (see pseudocode 2.9) with cost $O(n \log n)$. However, we showed that $\mathbb{V}_{rla}[C_T]$ can be computed exactly with a $O(n)$-time algorithm (see pseudocode 4.4).

The evidence we provide consists on analysing the correlation. In order to understand better the nature of this correlation we fit a linear model motivated by the graphical representation of the z-scores standardisation of D and C. Mathematically speaking, besides correlation, we analyse the values of a (slope) and b (intercept) in the linear model

$$C_z = a \cdot D_z + b, \tag{5.4}$$

both computed using the Theil-Sen estimator, a median-based linear model regression method, devised by Henri Theil [35] and independently by Pranab K. Sen [31]. We also analyse the values of Pearson's and Kendall's correlation coefficients. In order to perform both tasks, we processed every language's treebank available in the datasets by computing the necessary values in equations 5.2 and 5.3 so as to obtain D_z and C_z. For every sentence in a treebank, we computed $D_T(\pi)$ and $C_T(\pi)$ using the implicit linear arrangement π built with the labelling that the sentences carry in the treebanks (the order of the vertices in the treebanks is, in fact, the order of the words in the sentence which we can use to define π). The values $\mathbb{E}_{rla}[D_T]$ and $\mathbb{V}_{rla}[D_T]$ are computed using equations 1.8 and 1.9 respectively. The value $\mathbb{E}_{rla}[C_T]$ is calculated using equation 1.4, and $\mathbb{V}_{rla}[C_T]$ is calculated using one of the algorithms devised in this work (since we are dealing with trees, we used the specialised algorithm for trees, in pseudocode 4.4). Notice that for some trees we have $\mathbb{V}_{rla}[C_T] = 0$. For example, star trees, $\mathcal{S}_n$, have $\mathbb{V}_{rla}[C_{\mathcal{S}_n}] = 0$ [2]. The standardisation C_z in sentences whose tree is a star tree cannot be done since it is undefined. Similarly for those sentences whose tree T has $\mathbb{V}_{rla}[D_T] = 0$. For this reason, the results used in the coming explanations, and presented in the figures and tables to come, are for those sentences whose tree T has $\mathbb{V}_{rla}[C_T] \neq 0$ and $\mathbb{V}_{rla}[D_T] \neq 0$, which implies $n \geq 4$. In order to provide a complete analysis of the data obtained from the datasets we stratified the sentences in the treebanks. We did this in two levels. In the first, we classified them according to their length. Secondly, for a fixed length n, we classified the unlabelled trees of all sentences of n words into equivalence classes under graph isomorphism. In the figures to come, the number of different sentences of a certain length is indicated as "# obs", and the amount of different trees of the sentences of a given length as "# trees". Stratification is motivated by the likely presence of Simpson's paradox [33].

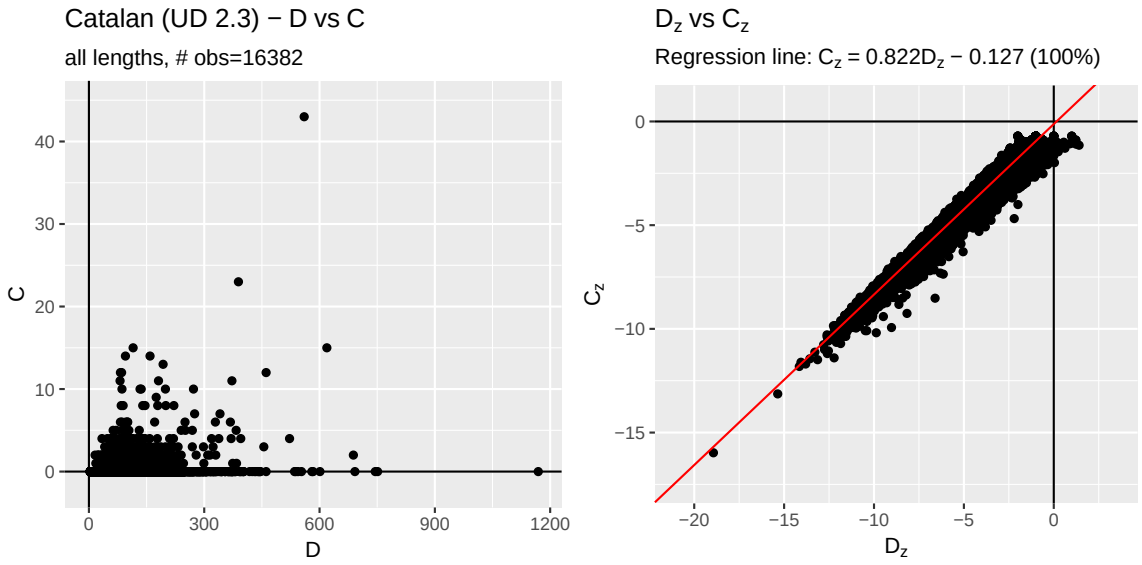


Figure 5.1: The relationship between D and C (left) and between D_z and C_z (right) in Catalan (*UD 2.3* dataset). The total amount of observations used is 16382. Each black dot in both graphs represent one tree in the dataset (one observation). In the graph to the right we find a clear regression line, estimated using the Theil-Sen estimator, giving $C_z = 0.822D_z - 0.127$. All the available observations were used to estimate it.

First of all, we discuss the linear models obtained using the Theil-Sen estimator without applying stratification, i.e., we analyse all sentences together regardless of their length. In figure 5.1(left) we show a graph of D vs C where the relationship between the variables is unclear at a first glance. However, as it is shown in figure 5.1(right), the z-score standardisation proves itself useful since we can see a noticeable linear relationship between D_z and C_z. The linear models found likewise for all

languages in the *UD 2.3*, *Prague* and *Stanford* datasets are presented, respectively, in tables 5.1 and 5.2 (the data in the *Prague* and *Stanford* datasets is given in the second table). Figure 5.2 shows the violin plots of the slopes and intercepts of the languages in the three datasets. Surprisingly, the slopes are concentrated around 0.75 in all languages and the values of the intercepts are negative most of the times. The exception is found in the *Prague* dataset (see figure 5.2(right)) where the values are concentrated around 0 and we even find a non-negligible amount of positive values. Since our analysis goes deeper than this, we decided not to dwell on this phenomenon.

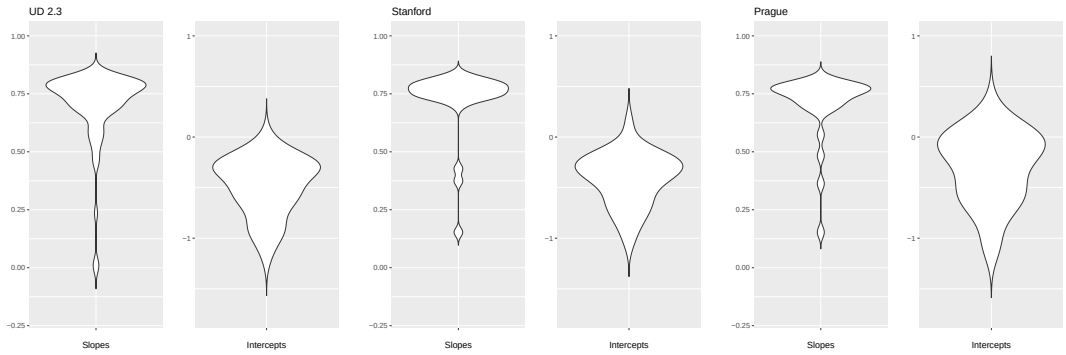
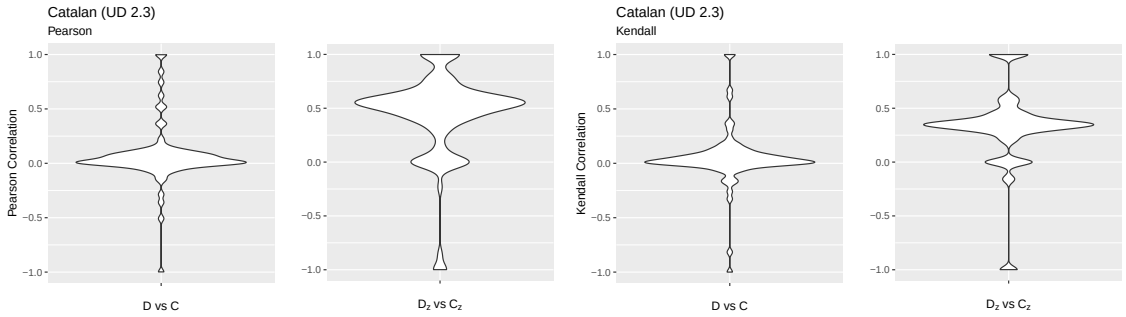


Figure 5.2: Violin plots of the slopes and intercepts of the linear models for D_z vs C_z for all languages in the datasets studied, indicated on top of each graph.

Now we aim at shedding light on the nature of the conclusion draw above, that there exists a strong correlation between D_z and C_z (as it can be seen in figures 5.1 and 5.2, and tables 5.1 and 5.2), through the lens of Simpson's paradox. Namely, we aim at figuring out whether the positive correlation is, in this context, due to the aggregation of uncorrelated data or the aggregation of negatively correlated data. In order to do this, we apply the first level of stratification. We grouped sentences by length and computed the correlation values of D vs C and D_z vs C_z for all of them within each group. These correlation values for Catalan of both Pearson's and Kendall's correlation coefficients, and for both D vs C and D_z vs C_z, for all the three datasets are shown using violin plots in figure 5.3. The results still point us in the direction that these two magnitudes are strongly positively correlated. The values of these correlations are also shown in figure 5.4(top) for Catalan, and we also give them for Japanese in figure 5.7(top), and for Dutch in figure 5.10(top). It is worth mentioning that the analysis for Catalan shows that the data extracted from some groups is uncorrelated. For example, the violin plots of D_z vs C_z (the second and fourth columns) of figure 5.3 show that there are several values of correlation around 0. This happens due to undersampling of long sentences. Table 5.3 shows the exact values used to make the top row of figure 5.3 and confirms that values close to 0 correspond to the longest sentences. In figures 5.4(bottom) and 5.7(bottom) we can see that the amount of sentences plummets as the sentence length increases, hence confirming undersampling.



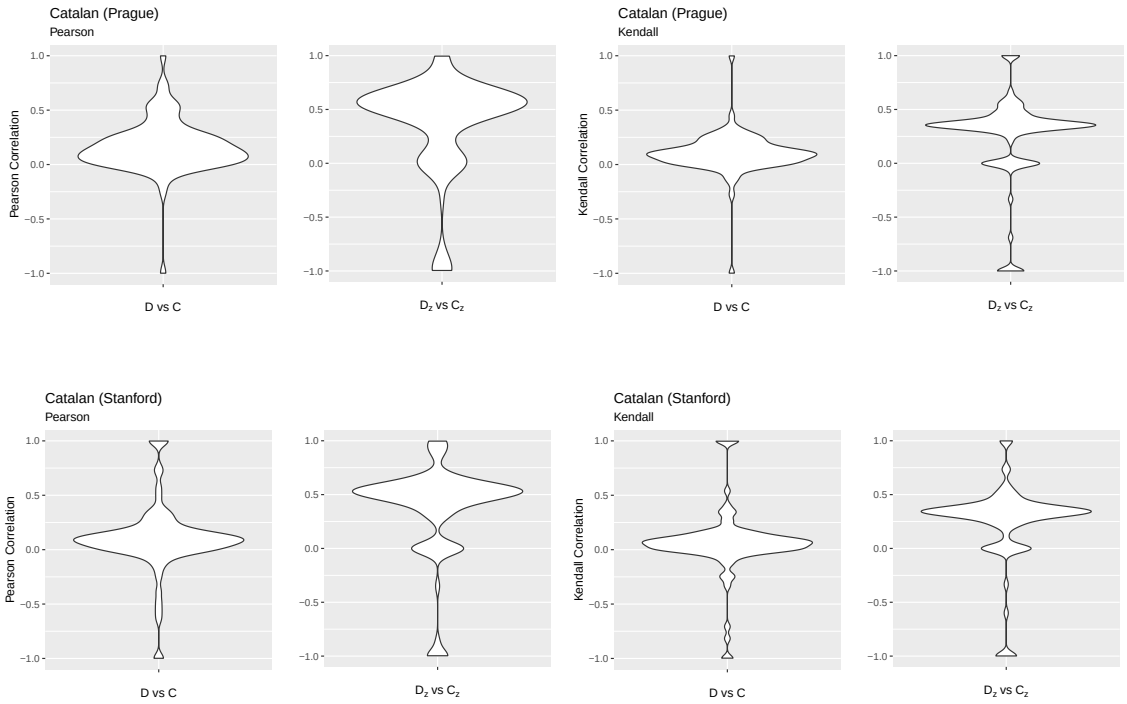


Figure 5.3: Pearon's and Kendall's correlation values for D vs C and D_z vs C_z among Catalan sentences of the same length. Figures in the top row use data from the *UD 2.3* dataset, with a total of 16382 observations. Figures in the middle row use data from the *Prague* dataset, with a total of 14556 observations. Figures in the bottom row use data from the *Stanford* dataset, with a total of 14520 observations.

Now we aim at investigating whether this positive linear relationship between D_z and C_z also exists when the same data is stratified at a deeper level. Here is where we use the classification of each sentence's tree into equivalence classes under isomorphism of graphs. More precisely, at every value of sentence length n in a treebank, we study the relationship among the classes of equivalence of the trees corresponding to the sentences of n words. In short, we want to make sure that for a given sentence length, the classification of the points (D_z, C_z) corresponding to the trees in the same class for a fixed length n do not produce flat lines (null slope). From a graph theoretical point of view, it is not so far-fetched to think that trees within the same equivalence class produce flat lines when D and C are standardised into D_z and C_z. All trees in one isomorphic class yield the same $\mathbb{E}_{rla}[C_T]$ and $\mathbb{V}_{rla}[C_T]$ [2], and the same $\mathbb{E}_{rla}[D_T]$ and $\mathbb{V}_{rla}[D_T]$ [7] (recall equations 1.8 and 1.9), therefore D_z and C_z have, respectively, only one free parameter D and C. Since C usually takes a null value ($C = 0$) the value on the y-axis is constant. With the value of D constantly changing the points are distributed on a flat line with intercept C_z, slope equal to 0 and values D_z.

If this was the case, there would be no correlation hence proving the existence of Simpson's paradox at the second level of stratification. Figures 5.5 and 5.8 show that this is the case in Catalan and Japanese, respectively (data extracted from the *UD 2.3* dataset). There we can see, in the right-most column, that trees belonging to the same isomorphic class (the groups of dots painted with the same colour) form flat lines which, when put all together, make the Theil-Sen estimator compute a non-null slope to fit this data. This shows the presence of Simpson's paradox. For length $n = 4$ (first row) we have a single class of tree for which $\mathbb{V}_{rla}[C] \neq 0$, the linear tree, making the Theil-Sen estimator obtain a line of slope 0, as expected. For $n = 5$ (second row) we have two classes of trees, linear and quasi-star trees, where, for each class, we find a null-slope. The aggregation of the two now makes the estimator produce a non-null slope. The same phenomenon occurs in $n = 6$ and in $n = 7$ but

only in Catalan. In Japanese, however, this does not happen (see figures 5.8(third row) and 5.8(fourth row)) since we have been able to spot (very few) points corresponding to trees of some class that are not aligned with the points of the other trees within the same class. By "very few" we mean that the amount of such trees (not sentences) not forming flat lines is negligible. Therefore, we should not consider it significant. In Dutch (see figure 5.11), however, the situation seems to have been completely reversed since there are more than just "very few" unaligned points. It can be appreciated very easily, in figure 5.11, that most of the data points corresponding the same type of trees are not aligned in Dutch for short sentences.

We have shown the presence of Simpson's paradox for sentences of short length in two languages, Catalan and Japanese. We have also shown that it is not present in Dutch, in sentences of short length. Therefore, it seems that the correlation between D_z and C_z may result mostly from a lucky superposition of flat lines in Catalan and Japanese, but not in Dutch.

However, we have to be cautious and check if this phenomenon can be observed in all sentence lengths. Unfortunately, on the one hand, we lack the necessary data to disprove Simpson's paradox in sufficiently long sentences. But, on the other hand, we can not confirm that the same phenomenon occurs in longer sentences. In other words, we are not able to make any type of decision, neither in favour or against it. This is due to undersampling, namely due to the simple fact that the longer the sentence the more likely it is to yield a tree that is the only representative of its isomorphic class. Put differently, for sufficiently long sentences, the number of sentences matches the number of classes of trees, i.e. each kind of tree is represented by only one sentence. This can be explained by simply analysing the amount of unlabelled free trees of a given number of vertices n. Otter [25] gave the growth of the number of unlabelled free trees $t(n)$ as a function of the number of vertices

$$t(n) \sim K\alpha^n n^{-5/2}, \quad \text{as } n \to \infty,$$

where $K \approx 0.53494...$ and $\alpha \approx 2.95576....$ This means that there is an exponential growth of this number of trees and hence giving a theoretical intuition in favour of this claim. Evidently, not all trees can be actual sentences, e.g. a sentence whose tree is isomorphic to a linear tree of 50 vertices is quite unlikely. Figures 5.4(bottom), 5.7(bottom) and 5.10(bottom), which show the number of classes of trees and the amount of sentences, both for all lengths up to 80 words in Catalan, Japanese[2] and Dutch, back this up: the lines depicting the number of classes of trees for each sentence length and the amount of trees of that length coincide almost perfectly (the precise values in table 5.3 show that there are slight differences in Catalan) for $n \geq 20$. Without such data, we are unable of completely acknowledging the presence of Simpson's paradox in longer sentences, or disprove it.

As a final brief, we have shed light on the nature of the correlation through the lens of Simpson's paradox. We have seen that the source of the correlation may depend on the language such as Catalan and Japanese where we have flat lines for specific trees, whereas in Dutch these flat lines disappear. Moreover, we have shown that we can not rule out Simpson's paradox in longer sentences due to undersampling. This is not an issue since correlation still exists since we can observe that the cloud of points corresponding to such sentences still follow the hypothesis, i.e., they are positively correlated at the first level of stratification, shown in figures 5.6, 5.9 and 5.12 for Catalan, Japanese[3] and Dutch, respectively. And we wonder, what is the correlation between D_z and C_z when the data is stratified at the "tree-type" level? Although we tried to answer the last question using linear trees and quasi-star trees, we were not able to obtain meaningful results due to their scarcity. And, unfortunately, this question had to be left unanswered.

[2] One reason we chose to analyse Japanese is because its treebank is much larger than Catalan's and Dutch's: figure 5.7(bottom) shows that there are, approximately, more than 16000 sentences of up to 30 words in the Japanese's treebank, whereas Catalan's treebank has slightly less than 16500 sentences in the whole treebank and Dutch's has around 17500.

[3] We would like to point out that the conclusions drawn using the data extracted for Japanese from the *UD 2.3* dataset can not be drawn so readily when using the *Prague* dataset. In this dataset the annotations of sentences were made in a way that crossings were somehow banned, yielding a constant $C = 0$ in practically all sentences [12] (we found exactly one case for which $C = 1$).

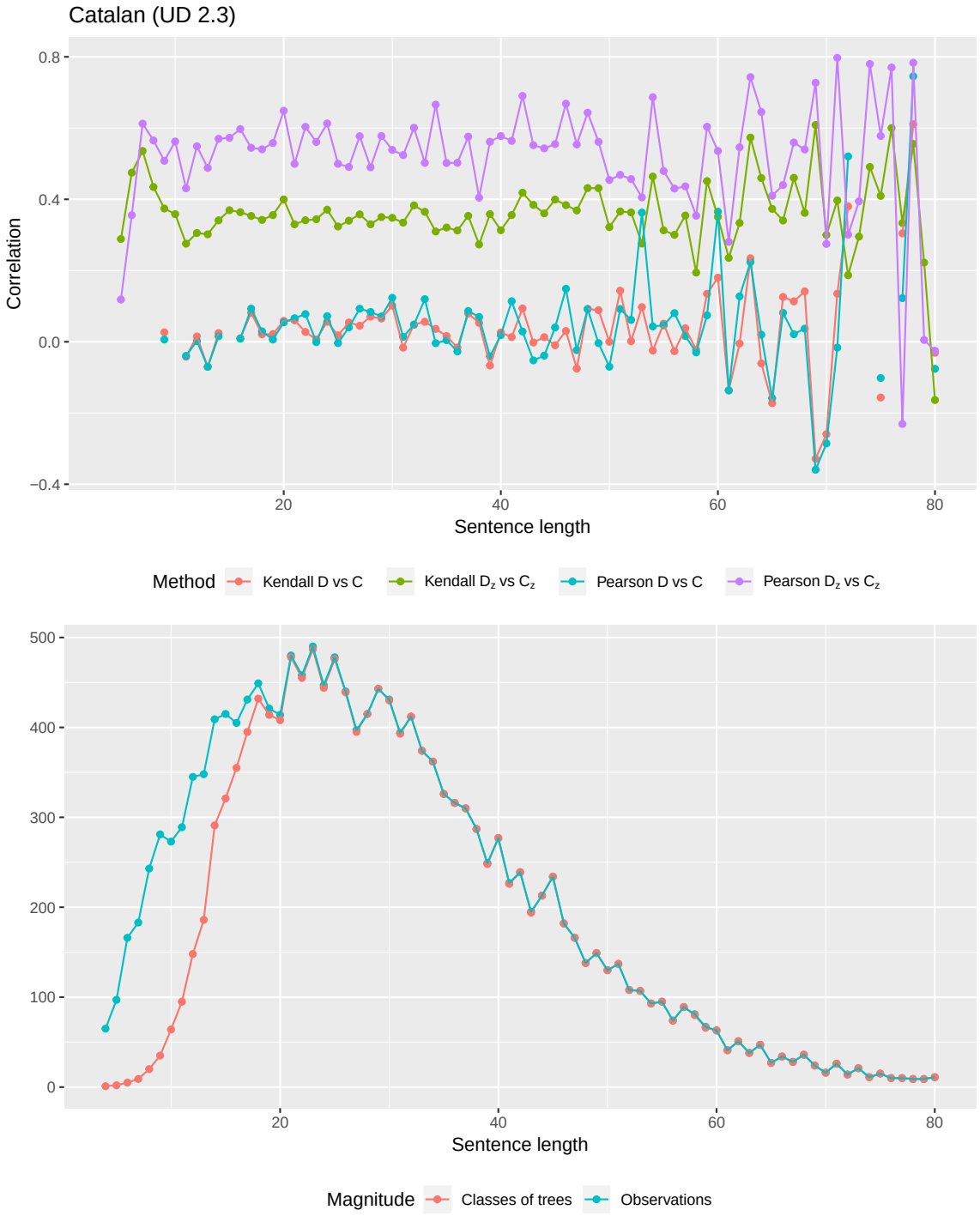


Figure 5.4: Top: correlation values in table 5.3 for all sentence lengths $n \leq 80$. Longer sentences are too scarce to provide meaningful results. Discontinuities are due to the fact the correlation is not defined according to the function in the *R* package that we used to calculate the correlation. Bottom: in red is shown the amount of different trees for each length, and in blue is shown the amount of sentences of length for each length.

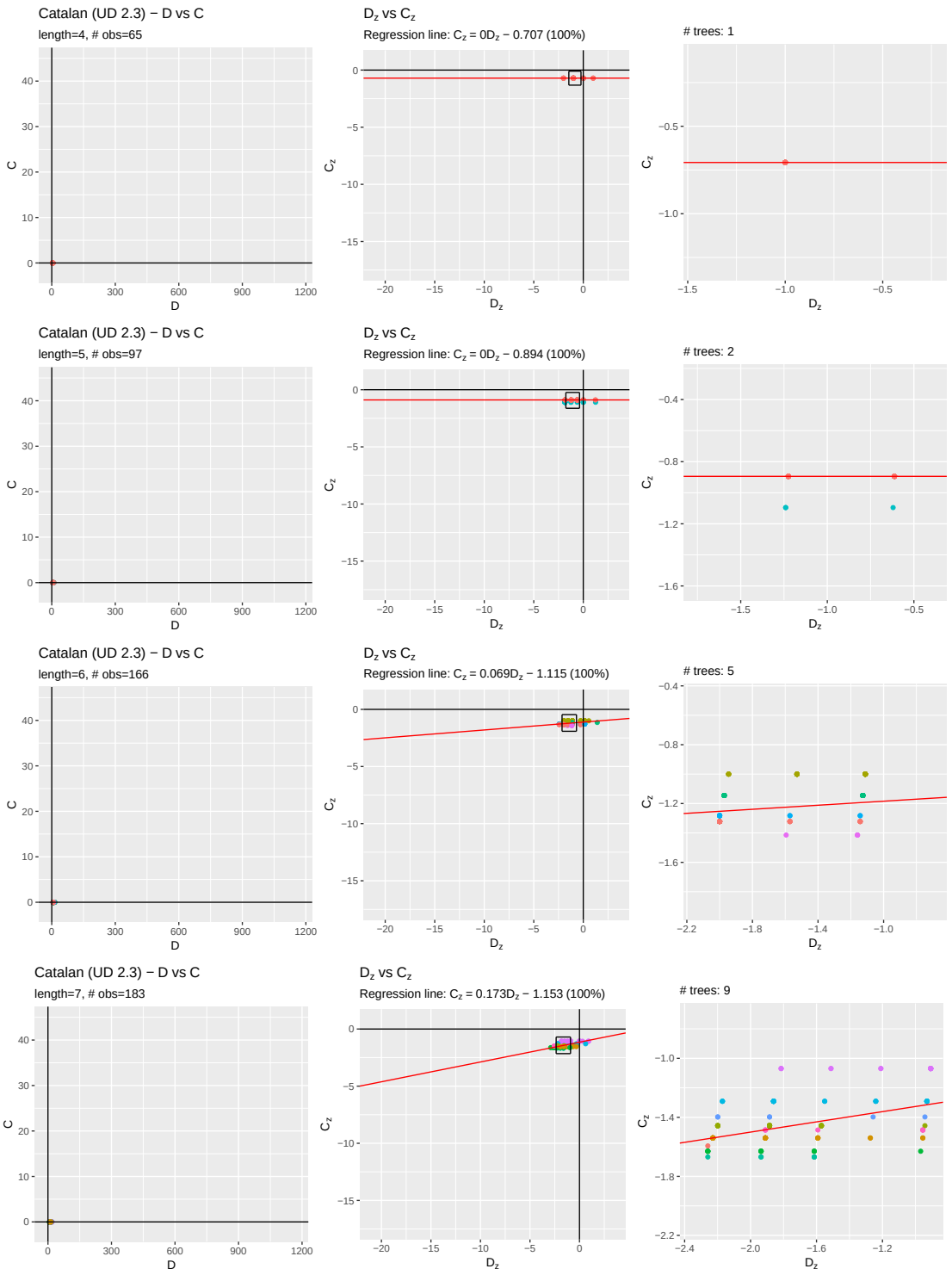


Figure 5.5: From top to bottom, the linear models for all sentences of lengths 4, 5, 6 and 7 in Catalan (dataset *UD 2.3*). Left Left column: D vs C. Middle column D_z vs C_z. Right column: a close-up of the region marked in the plot in the middle with a black rectangle. Dots are now painted using colours for each equivalence class of isomorphism for trees. The amount of different trees, is indicated at the top of the graph to the right.

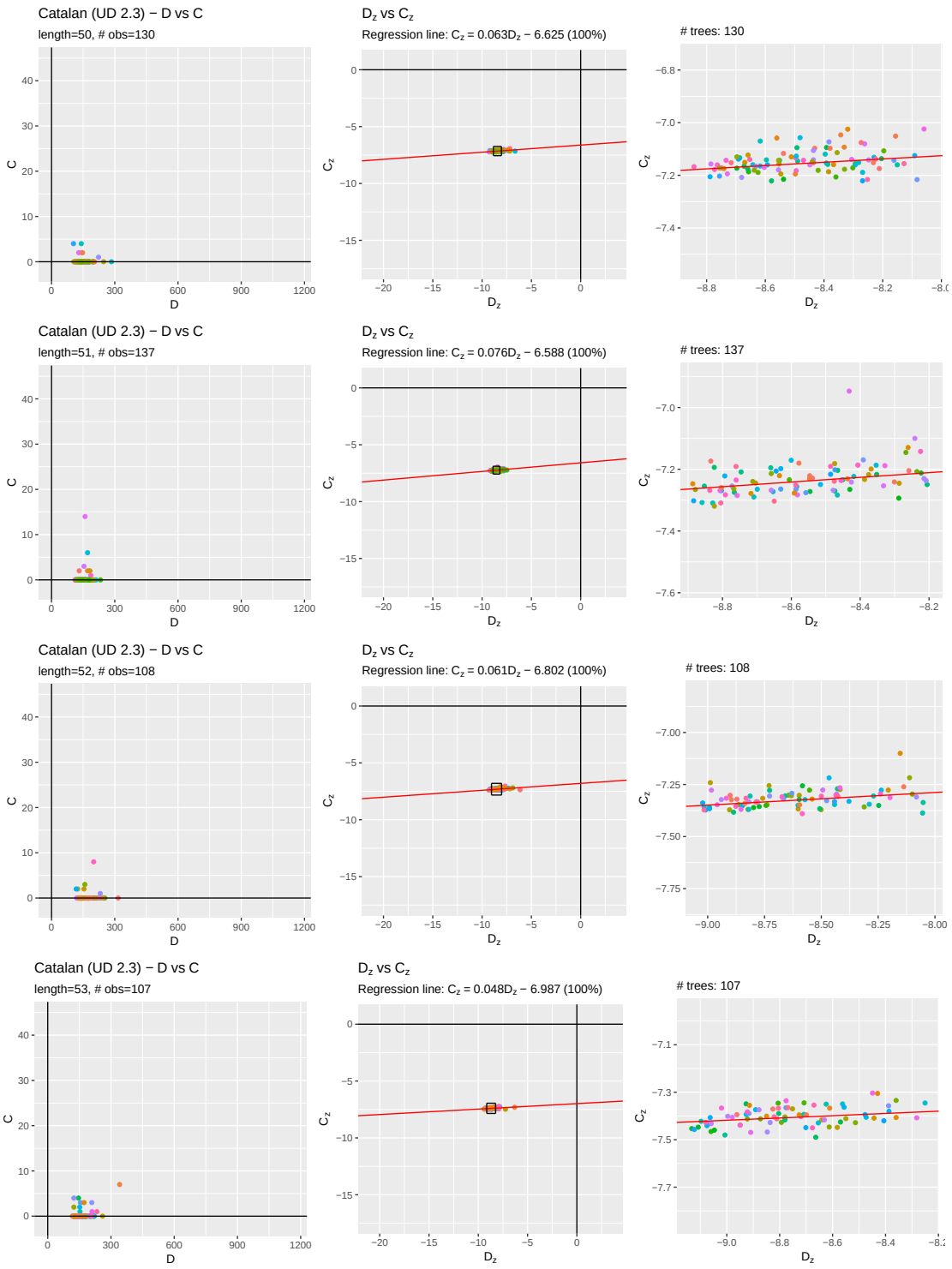


Figure 5.6: From top to bottom, the linear models for all sentences of lengths 50, 51, 52 and 53 in Catalan (dataset *UD 2.3*). Left Left column: D vs C. Middle column D_z vs C_z. Right column: a close-up of the region marked in the plot in the middle with a black rectangle. Dots are now painted using colours for each equivalence class of isomorphism for trees. The amount of different trees, is indicated at the top of the graph to the right.

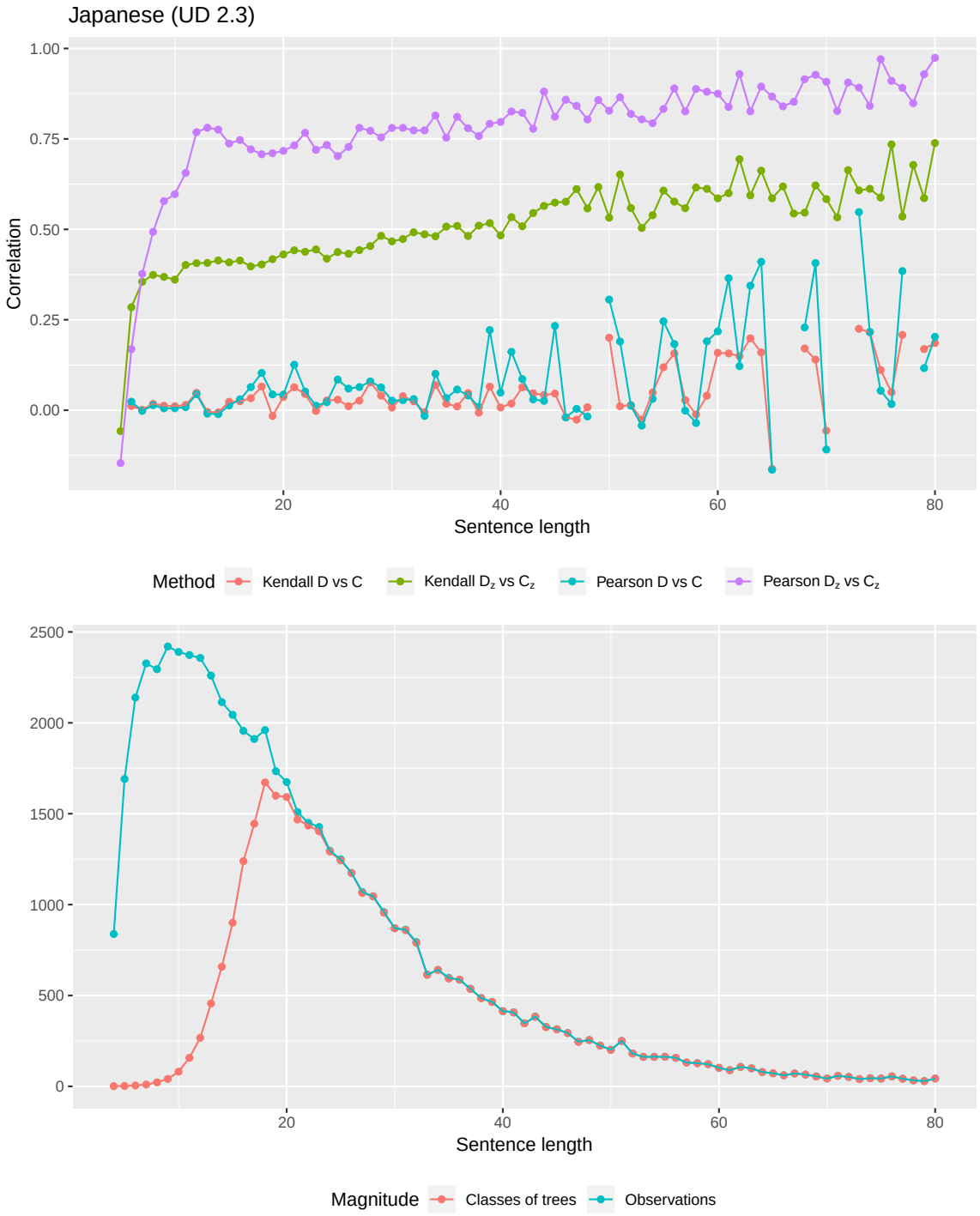


Figure 5.7: Top: correlation values for all sentences lengths of up to 80 words in Japanese (dataset *UD 2.3*). Longer sentences are too scarce to provide meaningful results. Discontinuities are due to the fact the correlation is not defined according to the function in the *R* package that we used to calculate the correlation. Bottom: in red is shown the amount of different trees for each length, and in blue is shown the amount of sentences of length for each length.

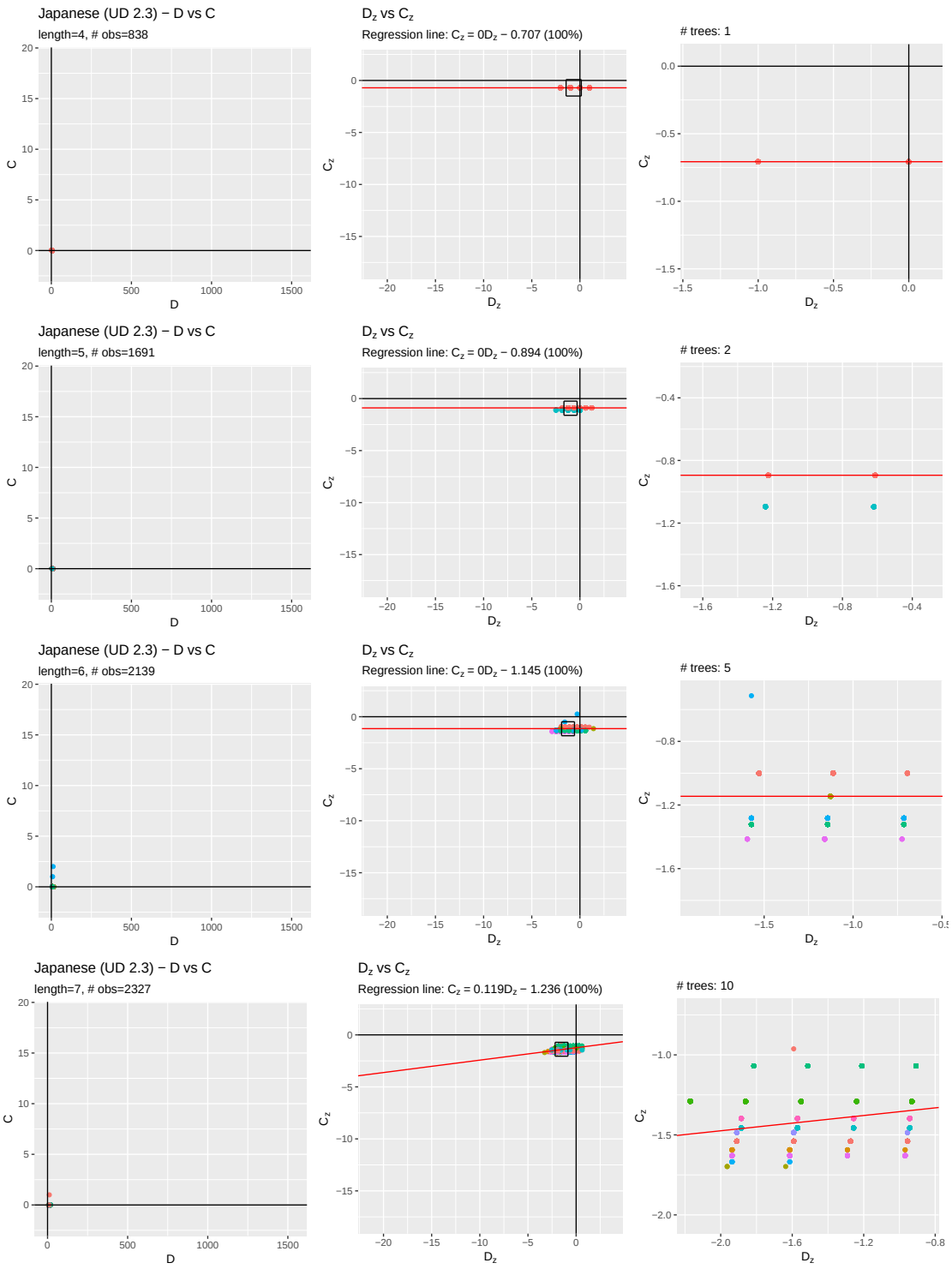


Figure 5.8: From top to bottom, the linear models for all sentences of lengths 4, 5, 6 and 7 in Japanese (dataset *UD 2.3*). Left column: D vs C. Middle column D_z vs C_z. Right column: a close-up of the region marked in the plot in the middle with a black rectangle. Dots are now painted using colours for each equivalence class of isomorphism for trees. The amount of different trees, is indicated at the top of the graph to the right.

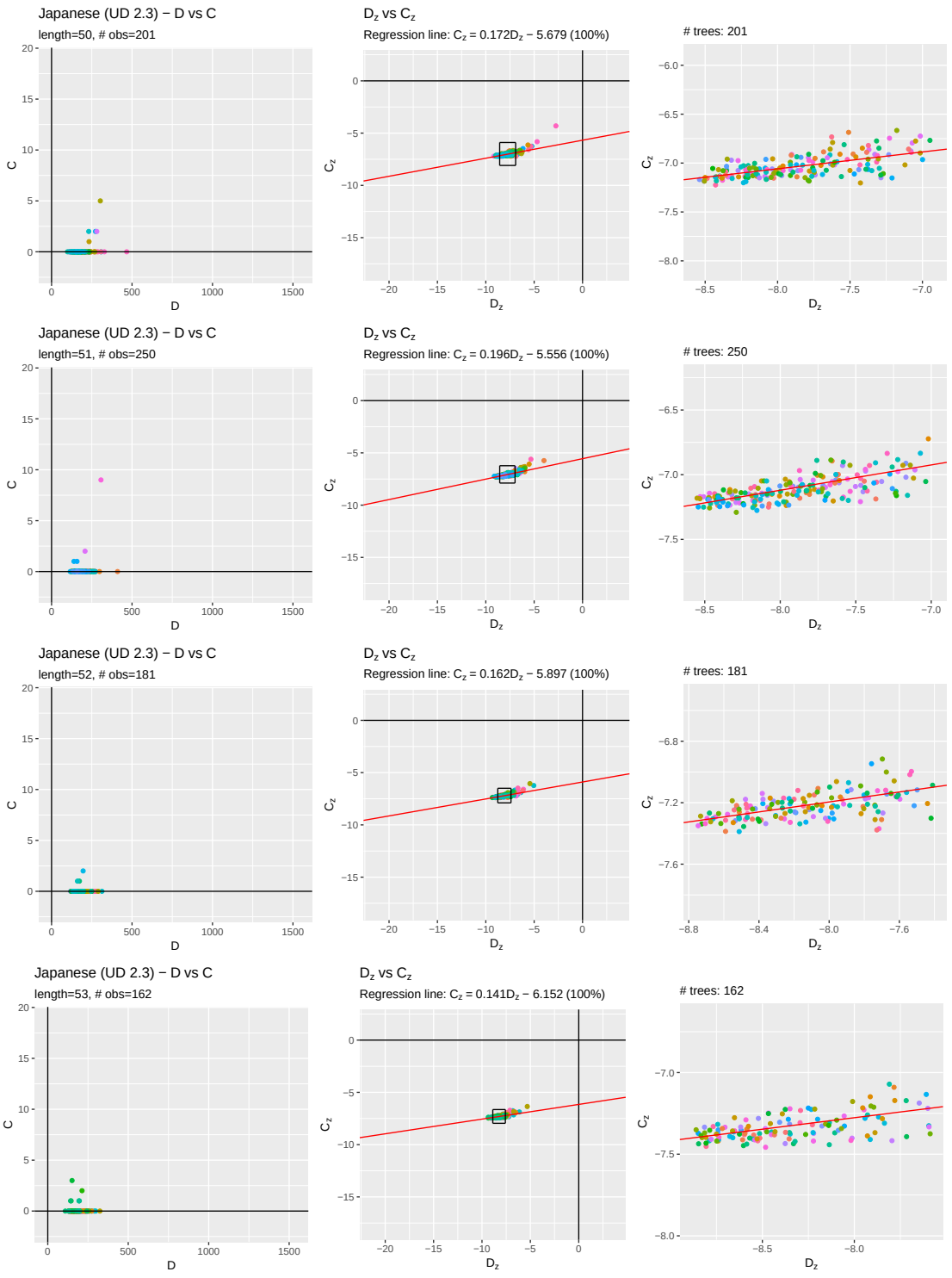


Figure 5.9: From top to bottom, the linear models for all sentences of lengths 50, 51, 52 and 53 in Japanese (dataset *UD 2.3*). Left Left column: D vs C. Middle column D_z vs C_z. Right column: a close-up of the region marked in the plot in the middle with a black rectangle. Dots are now painted using colours for each equivalence class of isomorphism for trees. The amount of different trees, is indicated at the top of the graph to the right.

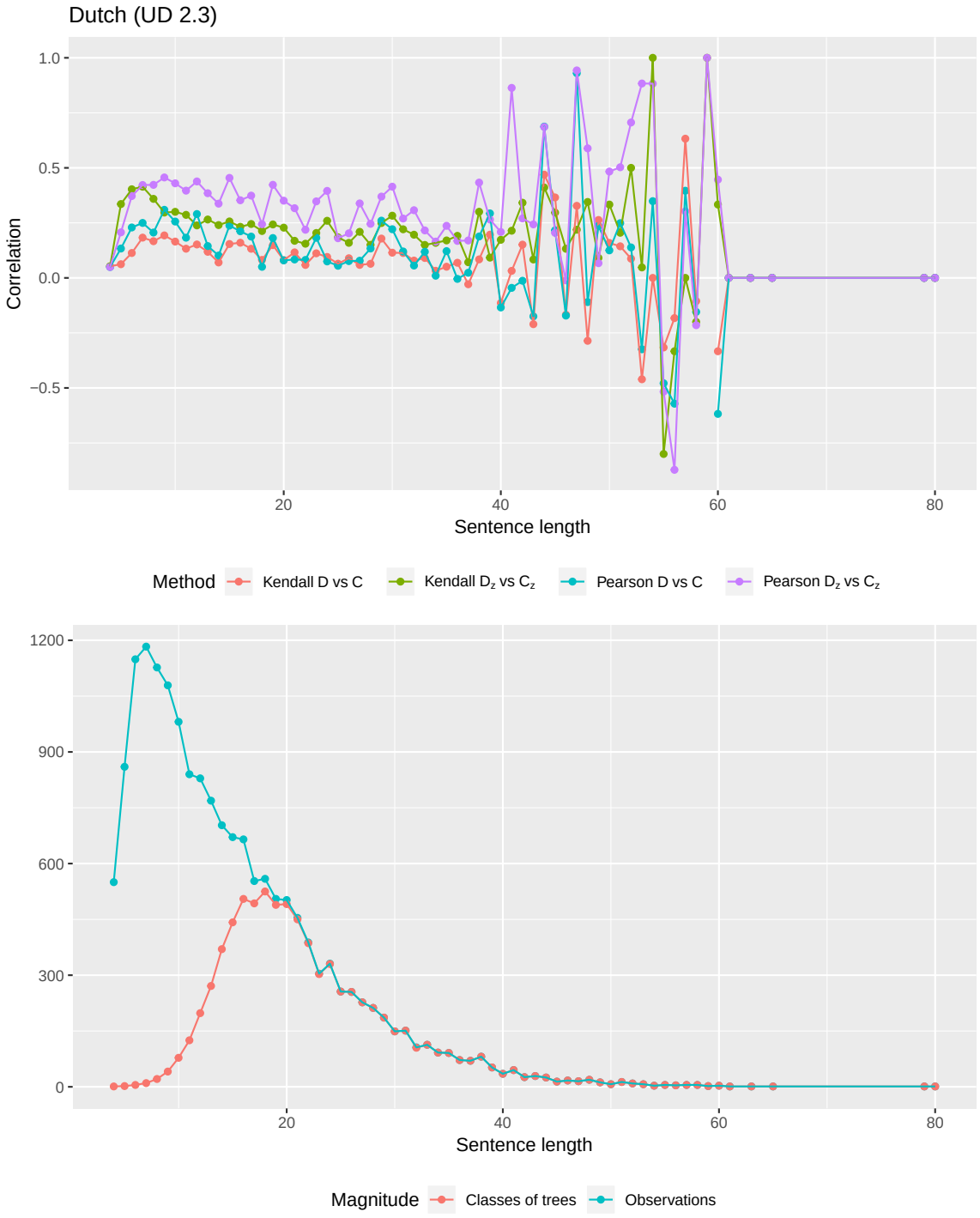


Figure 5.10: Top: correlation values for all sentences lengths of up to 80 words in Dutch (dataset *UD 2.3*). Longer sentences are too scarce to provide meaningful results. Discontinuities are due to the fact the correlation is not defined according to the function in the *R* package that we used to calculate the correlation. Bottom: in red is shown the amount of different trees for each length, and in blue is shown the amount of sentences of length for each length.

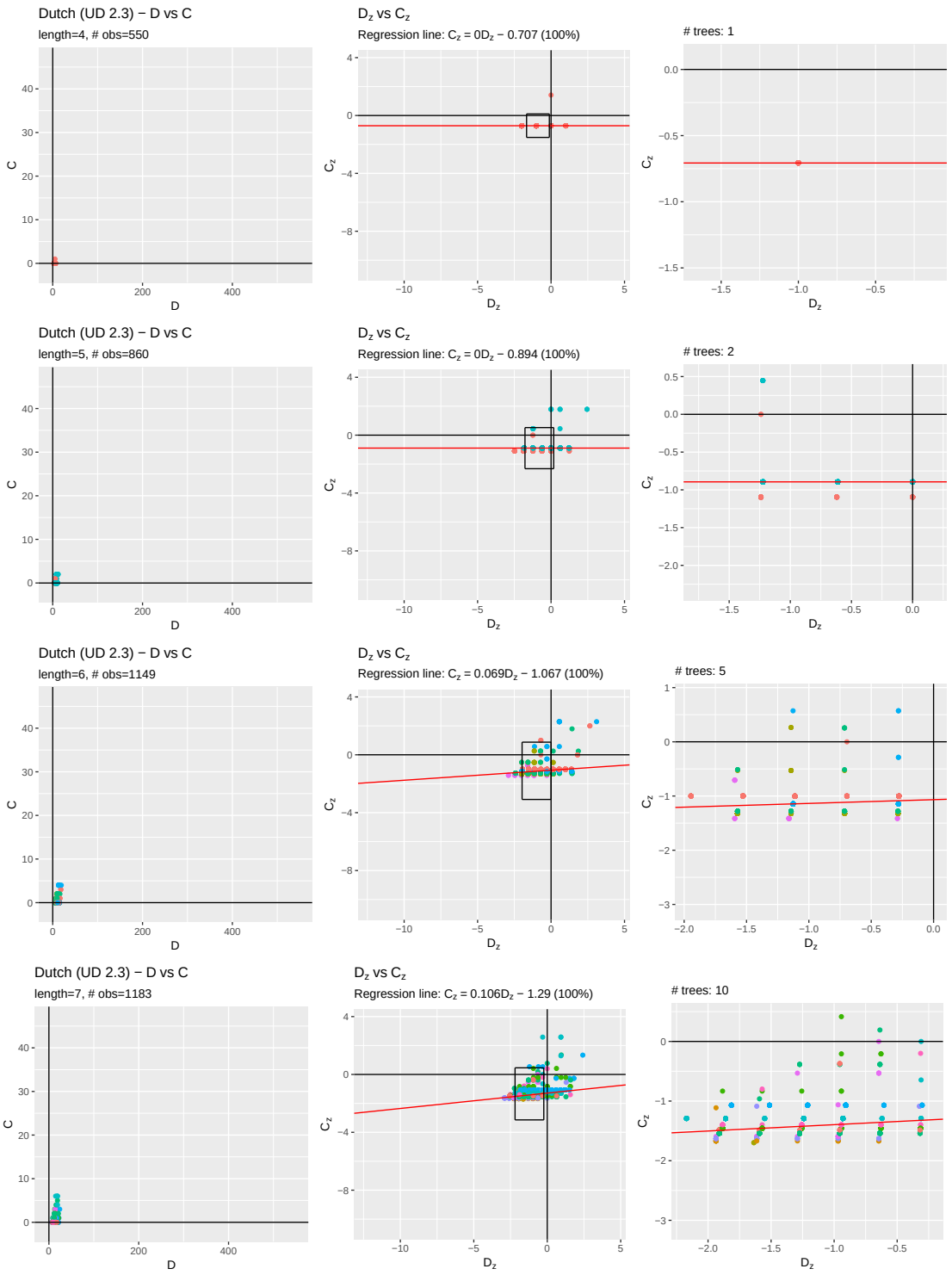


Figure 5.11: From top to bottom, the linear models for all sentences of lengths 4, 5, 6 and 7 in Dutch (dataset *UD 2.3*). Left column: Left column: D vs C. Middle column D_z vs C_z. Right column: a close-up of the region marked in the plot in the middle with a black rectangle. Dots are now painted using colours for each equivalence class of isomorphism for trees. The amount of different trees, is indicated at the top of the graph to the right.

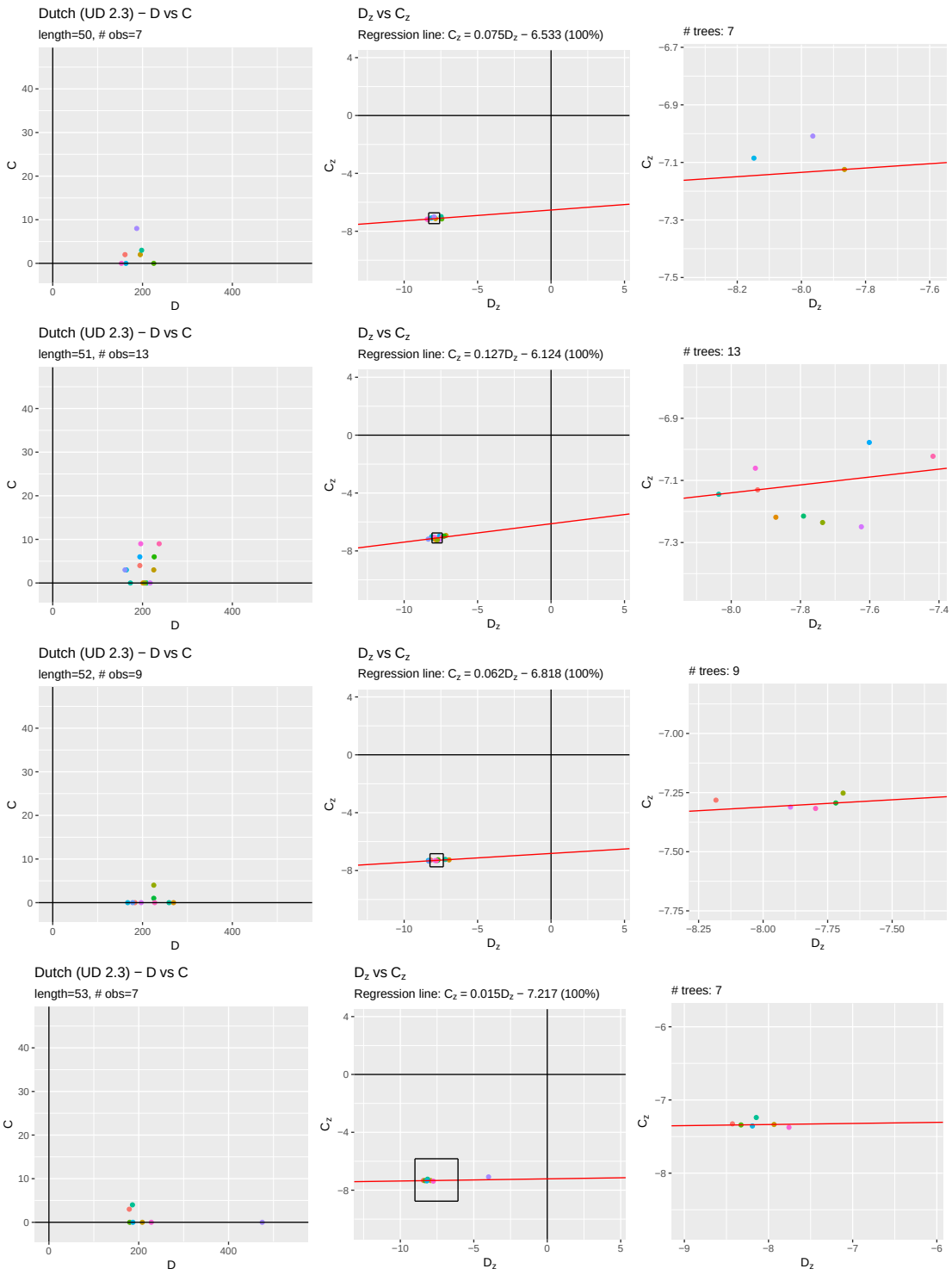


Figure 5.12: From top to bottom, the linear models for all sentences of lengths 50, 51, 52 and 53 in Dutch (dataset *UD 2.3*). Left column: D vs C. Middle column D_z vs C_z. Right column: a close-up of the region marked in the plot in the middle with a black rectangle. Dots are now painted using colours for each equivalence class of isomorphism for trees. The amount of different trees, is indicated at the top of the graph to the right.

Universal Dependecies 2.3 (1/3)				
Language	# Obs	% data	Slope	Intercept
Afrikaans	1923	100.000	0.700	-1.096
Akkadian	97	100.000	0.751	-0.628
Amharic	931	100.000	0.654	-0.379
Arabic	27171	73.608	0.818	-0.059
Bambara	898	100.000	0.733	-0.423
Belarusian	388	100.000	0.799	-0.158
Breton	753	100.000	0.668	-0.612
Buryat	791	100.000	0.664	-0.558
Bulgarian	10050	100.000	0.753	-0.356
Catalan	16382	100.000	0.822	-0.127
Czech	112769	17.735	0.773	-0.331
Old Church Slavonic	4842	100.000	0.693	-0.509
Chinese	6989	100.000	0.786	-0.698
Coptic	832	100.000	0.829	-0.224
Danish	4898	100.000	0.794	-0.314
German	16186	100.000	0.712	-0.924
Greek	2399	100.000	0.806	-0.206
English	29099	68.731	0.794	-0.336
Estonian	26383	75.806	0.745	-0.463
Basque	8369	100.000	0.765	-0.278
Faroese	981	100.000	0.510	-0.712
Finnish	27668	72.286	0.722	-0.345
French	40987	48.796	0.819	-0.147
Old French	14507	100.000	0.711	-0.500
Irish	940	100.000	0.795	-0.250
Galician	4919	100.000	0.767	-0.405
Gothic	4466	100.000	0.710	-0.510
Ancient Greek	27596	72.474	0.739	-0.590
Hebrew	6070	100.000	0.800	-0.174
Hindi	17561	100.000	0.750	-0.910
Hindi English	1887	100.000	0.598	-0.971

Universal Dependecies 2.3 (2/3)				
Language	# Obs	% data	Slope	Intercept
Croatian	8722	100.000	0.793	-0.279
Upper Sorbian	630	100.000	0.691	-0.808
Hungarian	1768	100.000	0.813	-0.351
Armenian	979	100.000	0.819	-0.229
Indonesian	6450	100.000	0.815	0.050
Italian	22705	88.086	0.787	-0.322
Japanese	58840	33.990	0.833	-0.238
Kazakh	908	100.000	0.547	-0.890
Kurmanji	735	100.000	0.571	-1.239
Korean	32388	61.751	0.624	-0.916
Komi Zyrian	220	100.000	0.710	-0.455
Latin	35781	55.896	0.737	-0.580
Latvian	8454	100.000	0.778	-0.302
Lithuanian	256	100.000	0.731	-0.576
Marathi	344	100.000	0.540	-0.707
Maltese	1865	100.000	0.811	-0.074
Erzya	1157	100.000	0.679	-0.395
Dutch	17422	100.000	0.766	-0.609
Norwegian	33425	59.835	0.790	-0.292
Naija	677	100.000	0.803	-0.164
Persian	5752	100.000	0.773	-0.938
Polish	18954	100.000	0.586	-0.535
Portuguese	21609	92.554	0.806	-0.185
Romanian	18870	100.000	0.777	-0.334
Russian	64017	31.242	0.764	-0.299
Sanskrit	167	100.000	0.452	-1.084
Slovak	8049	100.000	0.674	-0.478
Slovenian	9171	100.000	0.770	-0.414
North Sami	2185	100.000	0.678	-0.287
Spanish	33938	58.931	0.817	-0.114
Serbian	3855	100.000	0.793	-0.251

Universal Dependecies 2.3 (3/3)				
Language	# Obs	% data	Slope	Intercept
Swedish	10652	100.000	0.799	-0.295
Swedish Sign Language	157	100.000	0.467	-0.830
Tamil	584	100.000	0.698	-0.574
Telugu	508	100.000	0.019	-0.872
Tagalog	32	100.000	0.000	-1.095
Thai	994	100.000	0.782	-0.265
Turkish	5095	100.000	0.710	-0.707
Cantonese	378	100.000	0.689	-0.593
Uyghur	3033	100.000	0.591	-0.965
Ukrainian	5965	100.000	0.775	-0.279
Urdu	5121	100.000	0.801	-0.721
Vietnamese	2793	100.000	0.745	-0.251
Warlpiri	22	100.000	0.235	-0.674
Yoruba	99	100.000	0.808	-0.256

Table 5.1: Linear models of D_z vs C_z for all languages in the *UD 2.3* dataset, found using the Theil-Sen estimator. "# Obs" is the amount of sentences in the each language's treebank whose tree T is such that $\mathbb{V}_{rla}[C] \neq 0$ and $\mathbb{V}_{rla}[D] \neq 0$. "% data used" indicates the percentage of the data used to obtain the linear model. When the percentage is not 100%, the data was sampled u.a.r. (without replacement) due to our limitations of hardware resources.

Prague					Stanford				
Language	# Obs	% data	Slope	Intercept	Language	# Obs	% data	Slope	Intercept
Arabic	2248	100.000	0.817	0.314	Arabic	2280	100.000	0.835	0.128
Bengali	651	100.000	0.363	-1.040	Bengali	678	100.000	0.374	-1.027
Bulgarian	11947	100.000	0.771	0.046	Bulgarian	12119	100.000	0.762	-0.291
Catalan	14556	100.000	0.803	0.098	Catalan	14520	100.000	0.826	-0.113
Czech	70023	28.562	0.763	-0.175	Czech	74843	26.723	0.772	-0.329
Danish	4840	100.000	0.780	0.043	Danish	4894	100.000	0.784	-0.294
German	32443	61.647	0.761	-0.637	German	33492	59.716	0.796	-0.629
Greek	2543	100.000	0.795	0.005	Greek	2584	100.000	0.809	-0.170
English	18369	100.000	0.779	-0.146	English	18275	100.000	0.791	-0.360
Estonian	843	100.000	0.483	-0.631	Estonian	851	100.000	0.428	-0.703
Euskera	8717	100.000	0.764	-0.122	Euskera	9072	100.000	0.768	-0.232
Finnish	4011	100.000	0.754	-0.159	Finnish	4078	100.000	0.737	-0.341
Ancient Greek	16237	100.000	0.710	-0.505	Ancient Greek	18713	100.000	0.736	-0.447
Hindi	12334	100.000	0.718	-0.671	Hindi	12417	100.000	0.741	-0.895
Hungarian	5047	100.000	0.786	-0.413	Hungarian	6103	100.000	0.809	-0.451
Italian	2398	100.000	0.781	0.034	Italian	2502	100.000	0.791	-0.208
Japanese	4792	100.000	0.574	-0.481	Japanese	4614	100.000	0.726	-0.267
Latin	2833	100.000	0.711	-0.707	Latin	3036	100.000	0.743	-0.649
Dutch	11131	100.000	0.733	-0.322	Dutch	10974	100.000	0.770	-0.423
Persian	11632	100.000	0.677	-1.101	Persian	11579	100.000	0.749	-0.781
Portuguese	8596	100.000	0.787	0.102	Portuguese	8621	100.000	0.796	-0.183
Romanian	3193	100.000	0.719	-0.043	Romanian	3145	100.000	0.720	-0.208
Russian	31900	62.696	0.748	-0.189	Russian	31581	63.329	0.761	-0.324
Slovak	44297	45.150	0.750	-0.193	Slovak	47727	41.905	0.759	-0.354
Slovenian	1581	100.000	0.759	-0.197	Slovenian	1719	100.000	0.776	-0.424
Spanish	15424	100.000	0.791	0.050	Spanish	15354	100.000	0.814	-0.192
Swedish	10207	100.000	0.794	-0.028	Swedish	10714	100.000	0.802	-0.196
Tamil	585	100.000	0.678	-0.539	Tamil	584	100.000	0.695	-0.584
Telugu	373	100.000	0.153	-0.811	Telugu	429	100.000	0.153	-0.801
Turkish	3518	100.000	0.689	-0.473	Turkish	3862	100.000	0.729	-0.592

Table 5.2: Linear models of D_z vs C_z for all languages in the *Prague* (left) and *Stanford* (right) datasets, found using the Theil-Sen estimator. "# Obs" is the amount of sentences in the each language's treebank whose tree T is such that $\mathbb{V}_{rla}[C] \neq 0$ and $\mathbb{V}_{rla}[D] \neq 0$. "% data used" indicates the percentage of the data used to obtain the linear model. When the percentage is not 100%, the data was sampled u.a.r. (without replacement) due to our limitations of hardware resources.

Correlations in Catalan (UD 2.3) (1/4)						
			Pearson		Kendall	
n	# Obs	# trees	D vs C	D_z vs C_z	D vs C	D_z vs C_z
4	65	1				
5	97	2		0.1186		0.2885
6	166	5		0.3555		0.4747
7	183	9		0.6125		0.5358
8	243	20		0.5653		0.4346
9	281	35	0.0064	0.5081	0.0265	0.3741
10	273	64		0.5623		0.3583
11	289	95	-0.0395	0.4308	-0.0419	0.2755
12	345	148	0.002	0.5488	0.015	0.3054
13	348	186	-0.0701	0.4882	-0.0704	0.3017
14	409	291	0.0155	0.57	0.0244	0.341
15	415	321		0.5727		0.3693
16	405	355	0.0086	0.5972	0.01	0.3639
17	431	395	0.0932	0.5446	0.0816	0.3533
18	449	432	0.0293	0.5403	0.0209	0.3421
19	421	414	0.0064	0.5581	0.0215	0.3556
20	414	408	0.0544	0.6486	0.0588	0.3997
21	480	478	0.0661	0.4995	0.0602	0.3297
22	458	455	0.0778	0.6035	0.0277	0.3412
23	490	487	-0.0011	0.5609	0.0065	0.344
24	447	444	0.0719	0.6129	0.0561	0.3708
25	478	476	-0.0037	0.4996	0.018	0.3237
26	440	439	0.0404	0.4906	0.0541	0.34
27	397	395	0.0929	0.5772	0.0451	0.3577
28	415	415	0.0837	0.4899	0.0708	0.3299
29	443	443	0.0711	0.5778	0.0652	0.3501
30	431	430	0.1236	0.5386	0.1016	0.348
31	394	393	0.0137	0.5242	-0.0164	0.3345
32	412	412	0.0482	0.601	0.0482	0.383
33	374	374	0.1197	0.5023	0.0562	0.3649
34	362	362	-0.004	0.6658	0.0361	0.3097

Correlations in Catalan (UD 2.3) (2/4)						
			Pearson		Kendall	
n	# Obs	# trees	D vs C	D_z vs C_z	D vs C	D_z vs C_z
35	326	326	0.0044	0.502	0.0163	0.3207
36	316	316	-0.0272	0.5025	-0.0162	0.3125
37	310	310	0.0864	0.5758	0.0795	0.3535
38	287	287	0.0695	0.4046	0.0531	0.2732
39	249	248	-0.0408	0.5615	-0.0667	0.3584
40	277	277	0.0186	0.5777	0.0263	0.3131
41	227	226	0.1139	0.5641	0.0132	0.3558
42	239	239	0.0289	0.6903	0.0937	0.4185
43	195	194	-0.0523	0.5519	-0.0022	0.3844
44	213	213	-0.0389	0.5431	0.013	0.3605
45	234	234	0.0402	0.5549	-0.0098	0.3994
46	182	182	0.1489	0.6685	0.0303	0.3836
47	166	166	-0.0234	0.5538	-0.0751	0.3687
48	138	138	0.0915	0.6432	0.0922	0.4315
49	149	149	-0.0038	0.5614	0.0887	0.4313
50	130	130	-0.0701	0.4543	0	0.3217
51	137	137	0.092	0.4687	0.1434	0.366
52	108	108	0.0614	0.4568	0.0016	0.3632
53	107	107	0.3629	0.4053	0.0973	0.2757
54	93	93	0.043	0.6867	-0.0245	0.4639
55	95	95	0.0463	0.4794	0.0507	0.3129
56	74	74	0.0805	0.4302	-0.0264	0.3003
57	89	89	0.0162	0.4364	0.0379	0.3545
58	81	80	-0.0299	0.3538	-0.0224	0.1943
59	67	66	0.0742	0.6038	0.1347	0.4512
60	63	63	0.3652	0.5357	0.1801	0.3507
61	41	41	-0.1363	0.2806	-0.1369	0.2355
62	51	51	0.1276	0.546	-0.005	0.3336
63	38	38	0.2237	0.7431	0.2345	0.5733
64	47	47	0.0197	0.6452	-0.0609	0.4598
65	27	27	-0.1582	0.4099	-0.1724	0.3732

Correlations in Catalan (UD 2.3) (3/4)						
			Pearson		Kendall	
n	# Obs	# trees	D vs C	D_z vs C_z	D vs C	D_z vs C_z
66	34	34	0.0814	0.4401	0.1258	0.3405
67	28	28	0.0212	0.5596	0.1135	0.4603
68	36	36	0.0369	0.5398	0.1414	0.3619
69	24	24	-0.3592	0.7272	-0.3285	0.6087
70	16	16	-0.2855	0.2746	-0.2593	0.3
71	26	26	-0.0164	0.7973	0.1349	0.3969
72	14	14	0.5205	0.3009	0.3801	0.1868
73	21	21		0.3944		0.2952
74	11	11		0.7799		0.4909
75	15	15	-0.1017	0.578	-0.1565	0.4095
76	10	10		0.7702		0.6
77	10	10	0.1225	-0.2308	0.3043	0.3333
78	9	9	0.7454	0.7834	0.611	0.5556
79	9	9		0.0051		0.2222
80	11	11	-0.0759	-0.0252	-0.0309	-0.1636
81	9	9	0.8424	0.836	0.043	-0.1667
82	8	8	0.0427	0.3953	0.0714	0.2857
83	9	9	0.1179	-0.0111	0.3536	0.2222
84	5	5		0.5966		0.6
85	1	1	0	0	0	0
86	2	2		1		1
87	12	12	0.6222	0.6289	0.6742	0.5455
88	2	2		-1		-1

Correlations in Catalan (UD 2.3) (4/4)						
			Pearson		Kendall	
n	# Obs	# trees	D vs C	D_z vs C_z	D vs C	D_z vs C_z
89	3	3	-0.5076	0.9992	-0.8165	1
90	7	7		0.0629		-0.1429
91	2	2	1	1	1	1
92	4	4	0.1731	-0.8611	0.1826	-1
93	2	2	-1	1	-1	1
94	2	2		1		1
95	2	2		1		1
96	4	4	0.5167	0.1544	0.1826	0
97	2	2		1		1
98	4	4	0.0351	0.3878	-0.1826	0.3333
99	2	2		1		1
101	2	2		-1		-1
102	1	1	0	0	0	0
106	1	1	0	0	0	0
107	1	1	0	0	0	0
109	2	2		-1		-1
113	1	1	0	0	0	0
114	1	1	0	0	0	0
117	2	2	1	1	1	1
120	1	1	0	0	0	0
121	1	1	0	0	0	0
147	1	1	0	0	0	0
214	1	1	0	0	0	0

Table 5.3: Correlations for the Catalan language (UD 2.3 dataset). "# Obs" is the amount of sentences in the treebank for length. "# Trees" is the amount of different trees for each length. We found a total of 16382 observations with $\mathbb{V}_{rla}[C] \neq 0$ and $\mathbb{V}_{rla}[D] \neq 0$.

6 Discussion

As an introduction to studying the number of crossings in a linear arrangement of the vertices of a graph, we have provided efficient algorithms for its computation (section 2), summarised in table 1.1. Our efforts yielded two algorithms, one of them being particularly efficient for dense graphs (section 2.2), and the other for trees (section 2.3). Based on the insights given in [2] we studied statistical properties of the number of crossings in random graphs. In particular, we studied its expectation (section 3.1) and its variance (section 4.1). We also complemented existing work on the prediction of the number of crossings given knowledge on the length of the edges (section 3.2). Moreover, we have provided efficient and, more importantly, novel algorithms for the exact computation of $\mathbb{V}_{rla}[C_G]$, a problem that was not solved until now, to the best of our knowledge. We devised algorithms to compute $\mathbb{V}_{rla}[C_G]$ in general graphs (section 4.3.2), trees (section 4.3.3) and forests (section 4.3.4), the last two having a linear time complexity in the number of vertices, paramount to our study on the hypothesis on the scarcity of crossings in syntactic dependency trees being a side-effect of dependency length minimisation (section 5). These algorithms are summarised in table 1.2. We have been able to provide strong empirical support for this hypothesis.

Even though we knew about previous works on the computation of the number of cycles (by Alon *et. al.* [3]) and the number of paths of a certain length (by Movarraei [23]) that could have been used to derive the algorithms for the exact computation of $\mathbb{V}_{rla}[C_G]$, they were not suitable for our purposes, as discussed at the beginning of section 4.3.1. Because of this, we had to derive our own, a decision that took us through a long way of mathematical derivations (section 4.3.1), which has the clear disadvantage that we might not have derived those that take us to the most efficient algorithm for general graphs. Such expressions might exist, thus further research is necessary. Moreover, we tackled the problem of computing the exact value of $\mathbb{V}_{rla}[C_G]$ by analysing each summation of equation 4.17. This has two drawbacks. Firstly, deeper analyses might show that some of the terms involved in the computation of these summations may cancel each other out and yield a simpler algorithm. We made some attempts to achieve this, but we did not succeed. And secondly, by interpreting the terms as a whole we might find equivalent expressions, in a similar way that we did in propositions 4.3 and 4.4, that can be evaluated more easily.

Although we know that properties of trees can be exploited to derive more efficient algorithms, as it has been the case for the computation of the variance in section 4.3.3, we did not know how to exploit them to obtain asymptotically better algorithms to compute the number of crossings in linear arrangements of trees. Future work could investigate how to exploit these properties to obtain algorithms to solve the same problem with lower asymptotic costs.

We also wish we had been able to study lower bounds on the computation of $C_G(\pi)$, and on the computation of the exact value of $\mathbb{V}_{rla}[C_G]$ in general graphs. Moreover, even though the algorithms devised for the computation of $\mathbb{V}_{rla}[C_G]$ in time $O(n)$ in trees (section 4.3.3) and forests (section 4.3.4) seem to be optimal, we did not prove it. Along the same lines, future work could consist on studying lower bounds on the computation of the variance on general graphs and study even more efficient algorithms to solve this problem.

An important avenue for future work is the the development of statistical tests of significance for the number of crossings C. In previous research, Monte Carlo tests have been used to check if C is significantly low as expected in a random linear arrangement (see, for example, [12]) These tests use a Monte Carlo procedure to estimate a p-value. Fast statistical significance testing could be developed using Chebishev-like inequalities, e.g. one-sided Chebishev inequality also known as Cantelli's inequality [8, 26]. We could calculate an upper bound of the p-value using such inequality and then make a decision. If the upper bound is below the significance level then we would reject the null hypothesis. In case it was not, we would need to estimate the true p-value using a Monte Carlo procedure, and make a final decision based on that estimate. Such a procedure has been already outlined to check if D is significantly small [7].

The variance of C_G, $\mathbb{V}_{rla}[C_G]$, has a clear application on analysing linguistic networks and doing fast statistical significance tests. We think that there exist more applications of the variance applied to

combinatorics. In particular, it can be applied in solving the Mininum/Maximum Linear Arrangement problem. Just like the treewidth of a graph is used to indicate the difficulty of solving certain NP-complete problems, e.g., graph colouring, $\mathbb{V}_{rla}[C_G]$ might be a useful indicator of the difficulty of finding a linear arrangement that minimises (maximises) C (note that the minimisation problem is trivial in trees). If we say that two linear arrangements π_1 and π_2 are equivalent when they yield the same number of crossings, $C_G(\pi_1) = C_G(\pi_2)$, then $\mathbb{V}_{rla}[C_G]$ might be a measure of the amount of non-equivalent linear arrangements. The higher this amount, the more difficult it might be to find the optimal linear arrangement. The same rationale can be applied to D. Furthermore, future work might involve studying the expectation of the third moment of C_G, i.e. $\mathbb{E}_{rla}\left[C_G^3\right]$ so as to obtain more information on the upper bound of C_{min} on a particular graph, in a similar way it is done in [7, Section 7.3].

In sections 3.1 and 4.1 we analysed the expected value of $\mathbb{E}_{rla}[C_G]$ and $\mathbb{V}_{rla}[C_G]$ in Erdös-Rényi graphs. Future work should include similar analyses in other models of random graphs.

We could not find any practical application of our findings in section 3.2. Future work could involve the replications of the same analyses in [21], and and see if the same conclusions hold for unlabelled trees in the experiments in [6].

Finally, the data presented in section 5 provides strong empirical evidence that shows that there is a strong correlation between D and C, hence supporting the hypothesis that a low number of crossings in such trees is a side effect of the minimisation of the dependency length. However, as explained in that same section, deeper analyses are required to understand the real nature of the strong correlation between D_z and C_z beyond the Simpson's paradox [33] and the undersampling problems we have just unveiled. Moreover, not even an indisputable strong correlation between D and C can prove causality. This is left for future work.

A Linear Arrangement Library

All the algorithms presented in this work have been implemented in a C++ library (accessible online at [19] in the near future[4]) so as to provide the scientific community with implementations of these algorithms. They provide both exact rational and floating-point arithmetic. Since some of the potential users might not be used to programming with C++ (or C), the library has been interfaced to Python 3 using SWIG [34] (version 3.0.12).

Besides the algorithms presented in this work,

- Computation of the number of crossings. We implemented the algorithms described in sections 2.1, 2.2 and 2.3,
- Prediction of the number of crossings, as explained in section 3.2,
- Computation of $\mathbb{V}_{rla}[C_G]$ in general graphs (section 4.3.2), trees (section 4.3.3), and forests (section 4.3.4),

the library also includes algorithms for

- Exhaustive generation of labelled and unlabelled trees (labelled trees are generated using Prüfer codes [27], and unlabelled trees are generated using level sequences as described by Wright *et. al.*[37]),
- Random generation of labelled and unlabelled trees (labelled trees are again generated using Prüfer codes using a custom algorithm, and unlabelled trees are generated using the algorithm described by Wilf in [36]).

We also deemed important to implement other state of the art works, like the computation of the expectation and variance of the sum of the length of the edges (see equations 1.8 and 1.9), and to implement helper functions for input and output operations. These are not limited to reading graphs from a data file (in *edge list* format), these also include functions for parsing corpus of languages, in particular those that are made of syntactic dependency treebanks. These corpus are made of several treebank files which contain syntactic dependency trees. The amount of such trees depend on the language. The library offers two options regarding these treebanks: automatic processing of a whole treebank, which produces an output file with several properties of each tree in it (e.g., the expectation and variance of the number of crossings and of the sum of the length the edges), or manual processing of the treebank, which allows the user to iterate over each tree perform custom operations on each of them separately.

A.1 Protocol for testing

Since the algorithms presented in this work are not intuitive at all (see for example the algorithm for computing the variance in general graphs in pseudocode 4.2) and the implementation of other algorithms are quite prone to error (since they require the usage of data structures that we had to implement so as not to burden the user with dependencies of third-party libraries[5], e.g. the algorithm for computing the number of crossings in 2.3 for which we implemented our own AVL trees), we applied, from the beginning of this project, a validation protocol to ensure the correctness of the implementations.

In short, this validation protocol uses a script that orchestrates the testing of any algorithm with the help of a *tester* program. This *tester* is a piece of software that calls the functions in the library where the input parameters depend on the test being executed. In some tests, the result of these

[4] The library will be made publicly available when the most important parts of this thesis' work are submitted for publication in a peer-reviewed journal.

[5] Surely, some data structures, like AVL trees, are already implemented in many C++ libraries but this would have led to forcing the potential users of this library to have them installed in their computers.

functions is compared to some *ground truth*, values generated using a brute force algorithm. This *ground truth* is sometimes stored in disk, or is generated at runtime. For example, when testing the algorithm that computes the variance on general graphs we want to be sure that it produces the same output as a brute force algorithm, one that we know produces the correct answer. But if the latter takes hours to finish then this can not be done, so we compute these values once and store them in disk. On the other hand, we might want to test an algorithm on a huge amount of data, maybe generated at runtime, for which the correct answer can be obtained quickly with a brute force algorithm. In this case, we run the algorithm that is being tested and the brute force algorithm. Whenever the *tester* finds an inconsistency it issues an output message accordingly. The script that orchestrates the protocol captures these messages and stores them in a file appropriately.

In the following paragraphs we explain how we do this in more detail. The validation protocol that we designed is made up of three parts: (1) input test files that tell the *tester* what to test and how to do it, (2) *ground truth* files which store the information that takes too much time to execute every time we want to test the algorithms, and (3) the script that orchestrates the validation of each new algorithm.

Design of test files In order to make the tests as automatic as possible we first designed a simple format for test files. Each input test file has three keywords `TYPE`, `INPUT` and `BODY`, and they are written in this order. The first tells the *tester* what kind of test is to be run. The keyword is followed by a hyphenated sequence of strings indicating the type of test. The dashes allow an easier classification of the different type of tests. Figure A.1 shows an example of a test file. In that example the test type is `linear_arrangements-compute_C`, indicating a test related to linear arrangements and that we have to compute $C_G(\pi)$. Then follows the second keyword, `INPUT`, after which the *tester* will find the input test files. Since there can be many input files, we first write an integer s and then s strings each indicating the path to the files that the *tester* has to read. If the test needs as input a graph, the format (e.g., edge list) has to be specified after every file name. Finally, the third keyword, `BODY`, indicates the start of the contents of the tasks that the *tester* has to carry out. In the example, the contents of the `BODY` are: 1. the algorithm to be executed, in that case, the algorithm described in section 2.1, 2. the amount of linear arrangements to use to calculate $C_G(\pi)$, and 3. each of the linear arrangements, actually described as π^{-1}.

```
TYPE linear_arrangements-compute_C
INPUT 1 graphs/quasi-star-tree/005 edge-list
BODY
    dyn_prog
    3
    3 2 4 0 1
    3 4 1 2 0
    4 1 0 2 3
```

Figure A.1: Example of input test file.

Naming conventions of the input and *ground truth* files In order to know which should be the correct output of the *tester* (given an input test file) both input and *ground truth* files are named the same way: `test-****`, where the `****` is a 4-digit number identifying each file. The only constraint is that they need to be placed in different directories. Therefore, is no need to have a one-to-one correspondence between input and ground truth files. That is, two input files may be related to the same ground truth file, and even to have the same name. This allows us to run, automatically, different algorithms on the same input without having to replicate the *ground truth* files.

For example, consider an input test file where we want to test the computation of the variance of the number of crossings and that the input graphs are all trees. We can have a file named `test-0001`

stored in directory `inputs/properties/exp-var-C/general-formula/` and the contents of the `BODY` are such that the *tester* executes the algorithm to compute the variance in arbitrary graphs. Then, we can create another input file with similar contents, the only exception being the algorithm in the `BODY` field (make it so that, say, the *tester* executes the algorithm to compute the variance in trees), and store it in another directory, e.g. `inputs/properties/exp-var-C/trees/`, also with the same name. In this case, the output file is stored in `outputs/properties/exp-var-C/`.

Automatic testing An automatic testing protocol requires automatic checking of the outputs of each algorithm. We solved this issue in two ways.

On the one hand we stored the *ground truth* in the corresponding output files, generated with some brute force algorithm whose correctness is guaranteed (after thorough manual testing). Then, the *tester*'s output for an input test file is compared to the corresponding *ground truth* using the `diff` tool. This approach is very useful in those cases where the brute force algorithm is extremely slow to produce the correct output. For example, it was necessary to generate some *ground truth* to test the algorithms that computes the variance in general graphs, since the input graphs were quite large (around 100 vertices) and the brute force algorithm needs several hours to finish.

On the other hand, when the *ground truth* would be too large to store it in disk or the data produced could be easily produced by a brute force algorithm, we forced the test to execute the corresponding brute force algorithm. For example, the algorithm to compute the variance of C_G on trees (section 4.3.3) was tested not only on input graphs stored in disk[6], but also on trees generated in runtime. Since this type of tests do not produce an output, whenever the results by the brute force algorithm and by the algorithm being tested differ, an error message is issued. This error message is different for every type of test executed.

Automatic execution Obviously, comparing output files is a tedious task and also prone to error since we might forget to compare outputs produced by the *tester* and the *ground truth* files, and also to make sure that no error messages were produced. For this reason we designed a script that it is designed to execute the *tester* with all input test files inside a specified input directory and compare the outputs produced with the corresponding *ground truth* files in the specified output directory. Depending on the outcome of the *tester* the scripts issues a different message.

For example, it can execute all input test files in directory `inputs/lin-arrs/C/dyn-prog/` and compare each output in `outputs/lin-arrs/C/`. For every execution the *tester* makes sure that the outputs coincide, using the `diff` tool. If they do not, both the error output and standard output produced by the *tester* are stored in files whose names depend on the test executed. If the answers coincide but the *tester* issued an error message then the error output of the *tester* is moved to an appropriate file. In any case, the script outputs a message for each test executed. Figure A.2a shows the output of the script when the *tester*'s output coincides with the *ground truth*. Figure A.2b shows the script's output when the *tester*'s does not coincide with the *ground truth*. Finally, figure A.2c shows the script's messages when the *tester* issues an error message. An example of these error messages is found in figure A.2d.

```
Executing tests in inputs/properties/exp-var-C/general-formula/ (1/1)
    test-0000 (1/8) Ok
    test-0001 (2/8) Ok
    test-0002 (3/8) Ok
    test-0003 (4/8) Ok
    test-0004 (5/8) Ok
    test-0005 (6/8) Ok
    test-0006 (7/8) Ok
    test-0007 (8/8) Ok
```

(a) Successful test.

[6] In this case the output of the algorithm is compared to a *ground truth* stored in disk.

```
Executing tests in inputs/properties/exp-var-C/general-formula/ (1/1)
    test-0000 (1/8) Different outputs See result in .out.c++.properties.exp-var-C.general-formula.0000
    test-0001 (2/8) Different outputs See result in .out.c++.properties.exp-var-C.general-formula.0001
    test-0002 (3/8) Different outputs See result in .out.c++.properties.exp-var-C.general-formula.0002
    test-0003 (4/8) Different outputs See result in .out.c++.properties.exp-var-C.general-formula.0003
    test-0004 (5/8) Different outputs See result in .out.c++.properties.exp-var-C.general-formula.0004
    test-0005 (6/8) Different outputs See result in .out.c++.properties.exp-var-C.general-formula.0005
    test-0006 (7/8) Different outputs See result in .out.c++.properties.exp-var-C.general-formula.0006
    test-0007 (8/8) Different outputs See result in .out.c++.properties.exp-var-C.general-formula.0007
```

(b) Output of an algorithm is different from the *ground truth*.

```
Executing tests in inputs/linear-arrangements/C/dyn-prog (1/1)
    test-0000 (1/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0000
    test-0001 (2/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0001
    test-0002 (3/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0002
    test-0003 (4/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0003
    test-0004 (5/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0004
    test-0005 (6/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0005
    test-0006 (7/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0006
    test-0007 (8/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0007
    test-0008 (9/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0008
    test-0009 (10/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0009
    test-0010 (11/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0010
    test-0011 (12/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0011
    test-0012 (13/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0012
    test-0013 (14/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0013
    test-0014 (15/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0014
    test-0015 (16/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0015
    test-0016 (17/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0016
    test-0017 (18/18) Errors were produced See errors in .err.c++.linear-arrangements.C.dyn-prog.0017
```

(c) Tester program issued error messages.

```
Error [file exe_linarr_compute_C.cpp, function 'exe_linarr_compute_C', line 120]:
    Number of crossings do not coincide
        brute force: 1
        dyn_prog: 0
    For linear arrangement 0:
        [3,2,4,0,1]
**********************
Exiting with error type: test_error
```

(d) Example of error output of the *tester*.

Figure A.2: The three different outputs that the script can issue. In case of figures A.2b and A.2b, the algorithms were rigged to produce wrong answers.

Finally, since the library has been interface to python, the script was designed to be able execute the C++ version of the *tester*, or its Python 3 version, which loads the module that interfaces the C++ library to Python 3, as indicated via the input parameters of the script

- The C++ *tester* can be executed in either `debug` or `release` mode. In the former, the script is capable of using `valgrind` so that we can ensure that there are no memory leaks.

  ```
  ./test.sh --soft=c++ --release
  ./test.sh --soft=c++ --debug --valgrind
  ```

- The Python 3 version of the *tester* is executed similarly

  ```
  ./test.sh --soft=python
  ```

A.2 Validation tests

Since errors are usually made during the implementation process of any algorithm, however simple they might be, we implemented several automatic validation tests. In the following paragraphs we describe these tests.

Variance of C_T (section 4.3.3) We generated several "ground truth" files for several types of graphs: linear trees $\mathcal{L}_n$, quasi star graphs $\mathcal{Q}_n$, cycle graphs $\mathcal{C}_n$, one regular graphs $\mathbf{1}_m$, all for $2 \leq n \leq 100$, star trees $\mathcal{S}_n$ for $4 \leq n \leq 100$. These files contain $\mathbb{V}_{rla}[T]$ computed using a direct implementation of equation 4.4. The tests of the algorithm to compute $\mathbb{V}_{rla}[T]$ consisted on computing the variance for these graphs and compare the output stored in the ground truth files. Moreover, we also generated exhaustively and deterministically unlabelled free trees for several values of n. In this case we computed the variance with the algorithm for trees, for general graphs (section 4.3.2), and the direct implementation of equation 4.4, and checked that the results were equal. We use all $1 \leq n \leq 18$.

Variance of C_F on forests (section 4.3.4) Besides executing the algorithm on the same tests that we used on trees, we generated forests of random trees (not random forests), which consisted in a fixed number of labelled trees generated uniformly at random. The result of the algorithm was also compared against a direct implementation of equation 4.4 and against the result produced by the algorithm to compute the variance on general graphs (section 4.3.2). More precisely, each test consists on generating k forests of t random labelled trees, each of size n, and executing the three algorithms on the generated forest. Each test is to be repeated r times, usually $r = 10$. Smaller tests generate forests of $t = 2, 3$ random trees, each of n vertices for all $n \in [1, 10]$. Larger tests generate only $k = 4$ forests, this time of $t = 20, 30$ trees of $n = 7, 9$ vertices.

Variance of C_G on general graphs (section 4.3.2) Besides executing the algorithm on the same tests as the algorithm for trees is run on, we also did tests on complete graphs $\mathcal{K}_n$ for $2 \leq n \leq 20$, complete bipartite graphs $\mathcal{K}_{n_1,n_2}$ for $1 \leq n_1, n_2 \leq 9$, and random graphs $G_{n,p} \in \mathcal{G}_{n,p}$ for $10 \leq n \leq 50$ and $p = 0.0, 0.1, 0.2, \cdots, 1.0$. The results are compared against the ground truth generated with a direct implementation of equation 4.4 since it takes hours to finish for the largest random graphs (large n and large p).

Computation of $|\alpha(d_1, d_2)|$ and $|\beta(d_1, d_2)|$ (sections 3.2.1 and 3.2.2) This seems to be a case in which no automatic testing is required. However, due to the likelihood of this library being extended in the future, we also included tests for these algorithms. For $1 \leq n \leq 50$, we compute by means of brute force algorithms (see pseudocodes 3.1 and 3.2) and compare the results produced by the corresponding constant-time algorithms (see pseudocodes 3.3 and 3.5).

Algorithms for the computation $C_G(\pi)$ (section 2) We tested these algorithms using some of the graphs used for the tests of the algorithms of $\mathbb{V}_{rla}[T]$, i.e., $\mathcal{L}_n$, $\mathcal{Q}_n$, $\mathcal{C}_n$, ..., and $\mathbb{V}_{rla}[C_G]$, i.e., $\mathcal{K}_n$ for $2 \leq n \leq 20$, ... We generated a different amount of linear arrangements depending on the graph that the test uses. For example, one of the tests uses $\mathcal{Q}_5$ and executes the algorithm being tested on 20 linear arrangements. Other tests use the random graphs, e.g., one uses $G_{50,1/2}$ and 250 linear arrangements. All the linear arrangements were generated uniformly at random. Since the ground truth can be generated very easily, the result of the algorithm tested (see section 2) is compared against the result of a direct implementation of equation 1.3. Each algorithm is tested independently of the others.

Exact arithmetic Since the library provides exact arithmetic computation of the expectation and variance of the number of crossings (see equations 1.4 and 4.1), and of the sum of the length of the edges (see equations 1.8 and 1.9), we deemed important to test the custom wrapper on the GMP

library that was made for the library. The tests merely consist on several arithmetic expressions whose result is stored in a "ground truth" file and whose contents are ensured to be correct (by thorough manual inspection). This is done in case the interface is extended and/or modified to obtain faster implementations of these wrappers.

Other tests Other tests, of minor importance, were also implemented for the sake of completeness. For example, the computation of $\mathbb{E}_{rla}[D(G)]$ and $\mathbb{V}_{rla}[D(G)]$ (in equations 1.8 and 1.9) is not subject to direct changes. Therefore, once their implementation is correct, the results are not likely to change. However, we also included tests because they rely on other operations that might change.

References

[1] Georgy Adelson-Velsky and Evgenii Landis. "An algorithm for the organization of information". In: *Proceedings of the USSR Academy of Sciences*. 1962, pp. 263–266.

[2] L. Alemany-Puig and R. Ferrer i Cancho. "Edge crossings in random linear arrangements". In: *in prep* (2019).

[3] N. Alon, R. Yuster, and U. Zwick. "Finding and counting given length cycles". In: *Algorithmica* 17.3 (Mar. 1997), pp. 209–223.

[4] Frank Bernhart and Paul C Kainen. "The book thickness of a graph". In: *Journal of Combinatorial Theory, Series B* 27.3 (1979), pp. 320 –331. ISSN: 0095-8956.

[5] Béla Bollobás. *Modern Graph Theory*. 1st ed. Springer, 1998.

[6] R. Ferrer i Cancho. "A stronger null hypothesis for crossing dependencies". In: *CoRR* abs/1410.5485 (2014).

[7] Ramon Ferrer-i Cancho. "The sum of edge lengths in random linear arrangements". In: *Journal of Statistical Mechanics: Theory and Experiment* 2019 (May 2019), p. 053401. DOI: 10.1088/1742-5468/ab11e2.

[8] F. P. Cantelli. "Intorno ad un teorema fondamentale della teoria del rischio". In: *Bollettino dell'Associazione degli Attuari Italiani* 24 (1910), pp. 1–23.

[9] F.R.K. Chung. "On optimal linear arrangements of trees". In: *Computers & Mathematics with Applications* 10.1 (1984), pp. 43–60. ISSN: 0898-1221.

[10] Germinal Cocho et al. "Rank Diversity of Languages: Generic Behavior in Computational Linguistics". In: *PLOS ONE* 10.4 (Apr. 2015), pp. 1–12.

[11] J. L. Esteban and R. Ferrer-i-Cancho. "A correction on Shiloach's algorithm for minimum linear arrangement of trees". In: *SIAM Journal of Computing* 46 (3 2015), pp. 1146–1151.

[12] R. Ferrer-i-Cancho, C. Gómez-Rodríguez, and J. L. Esteban. "Are crossing dependencies really scarce?" In: *Physica A: Statistical Mechanics and its Applications* 493 (2018), pp. 311–329.

[13] Ramon Ferrer-i-Cancho and Carlos Gómez-Rodríguez. "Crossings as a side effect of dependency lengths". In: *Complexity* 21 (2016), pp. 320–328.

[14] M. R. Garey and D. S. Johnson. "Crossing number is NP-Complete". In: *SIAM Journal on Computing* 4 (3 1983), pp. 312–316. ISSN: 0196-5212.

[15] Branko Grünbaum. *Arrangements and Spreads*. Cbms Regional Conference Series in Mathematics 10. American Mathematical Society, 1972. ISBN: 9780821816592.

[16] Jan Hajič et al. *Prague Dependency Treebank 2.0*. CDROM CAT: LDC2006T01, ISBN 1-58563-370-4. Linguistic Data Consortium. Philadelphia, PA, USA, 2006.

[17] F. Harary and B. Manvel. "On the number of cycles in a graph". In: *Matematický časopis* 21.1 (1971), pp. 55–63.

[18] Robert Hochberg and Matthias Stallmann. "Optimal one-page tree embeddings in linear time". In: *Inf. Process. Lett.* 87 (July 2003), pp. 59–66. DOI: 10.1016/S0020-0190(03)00261-8.

[19] *Linear Arrangement Library*. https://github.com/lluisalemanypuig/linear-arrangement-library. Accessed: 2019-06-15.

[20] Marie-Catherine de Marneffe and Christopher D. Manning. "The Stanford Typed Dependencies Representation". In: *COLING 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*. Manchester, UK: COLING 2008 Organizing Committee, 2008, pp. 1–8. URL: http://aclweb.org/anthology/W08-1301.

[21] "Non-crossing Dependencies: Least Effort, Not Grammar". In: *Towards a Theoretical Framework for Analyzing Complex Linguistic Networks*. Ed. by Alexander Mehler et al. 1st ed. Springer, Berlin, Heidelberg, 2016. Chap. 3, pp. 203–231. ISBN: 978-3-662-47237-8.

[22] J. W. Moon. "On the Distribution of Crossings in Random Complete Graphs". In: *Journal of the Society for Industrial and Applied Mathematics* 13 (2 1965).

[23] N. Movarraei and M. Shikare. "On the number of paths of lengths 3 and 4 in a graph". In: *International Journal of Applied Mathematical Research* 3.2 (2014).

[24] J. Nivre et al. *Universal Dependencies 2.3*. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University. 2018. URL: http://hdl.handle.net/11234/1-2895.

[25] Richard Otter. "The Number of Trees". In: *Annals of Mathematics* 49.3 (1948), pp. 583–599. ISSN: 0003486X. URL: http://www.jstor.org/stable/1969046.

[26] M. Padulo and M. D. Guenov. "Worst-case robust design optimization under distributional assumptions". In: *International Journal for Numerical Methods in Engineering* 88.8 (2011), pp. 797–816.

[27] H. Prüfer. "Neuer Beweis eines Satzes über Permutationen". In: *Arch. Math. Phys* 27 (1918), pp. 742–744.

[28] Fan R. K. Chung, Frank Thomson Leighton, and Arnold Rosenberg. "Embedding Graphs in Books: A Layout Problem with Applications to VLSI Design". In: *Siam Journal on Algebraic and Discrete Methods* 8 (Jan. 1987).

[29] H. N. de Ridder et al. *Information System on Graph Classes and their Inclusions (ISGCI)*. http://www.graphclasses.org.

[30] G. Ringel. "Extremal problems in the theory of graphs". In: *Theory of Graphs and its Applications*. 1964, pp. 85–90.

[31] Pranab Kumar Sen. "Estimates of the Regression Coefficient Based on Kendall's Tau". In: *Journal of the American Statistical Association* 63.324 (1968), pp. 1379–1389. DOI: 10.1080/01621459.1968.10480934.

[32] Y. Shiloach. "A Minimum Linear Arrangement Algorithm for Undirected Trees". In: *SIAM Journal on Computing* 8.1 (1979), pp. 15–32.

[33] E. H. Simpson. "The Interpretation of Interaction in Contingency Tables". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 13.2 (1951), pp. 238–241. ISSN: 00359246. URL: http://www.jstor.org/stable/2984065.

[34] *SWIG*. http://swig.org/. Accessed: 2019-06-15.

[35] Henri Theil. "A Rank-Invariant Method of Linear and Polynomial Regression Analysis". In: *Proceedings of the Koninklijke Nederlandse Akademie Wetenschappen, Series A – Mathematical Sciences* 53 (Jan. 1950), pp. 386–392, 521. DOI: 10.1007/978-94-011-2546-8_20.

[36] Herbert S. Wilf. "The uniform selection of free trees". In: *Journal of Algorithms* 2 (2 1981), pp. 204–207. ISSN: 0196-6774.

[37] Robert Alan Wright et al. "Constant Time Generation of Free Trees". In: *SIAM Journal on Computing* 15 (May 1986), pp. 540–548.